Deterministic Synchronous Self-Stabilizing BFS Construction with Constant Space Complexity

Lélia Blin
Université Paris Cité, CNRS
IRIF, F-75013 Paris, France

Franck Petit
Sorbonne Université, CNRS
LIP6, Paris, France

Sébastien Tixeuil
Sorbonne Université, CNRS
LIP6, IUF, Paris, France

Abstract

In this paper, we resolve a long-standing open problem in self-stabilization asking whether it is possible to construct a spanning tree using *constant* memory per node in a synchronous semi-uniform networks, i.e., networks in which one node is distinguished. We design a synchronous self-stabilizing algorithm that constructs a breadth-first search (BFS) tree in any *anonymous* semi-uniform network using only a constant number of bits of memory per node. Crucially, our approach operates without any prior knowledge of global network parameters such as maximum degree, diameter, or number of nodes. In contrast to traditional self-stabilizing methods – such as pointer-to-neighbors, distance-to-root, or identifiers – that are unsuitable under strict memory constraints, our solution employs an innovative constant-space token dissemination mechanism. This mechanism effectively eliminates cycles and rectifies errors in the BFS structure, ensuring both correctness and memory efficiency. The proposed algorithm not only meets the stringent requirements of memory-constrained distributed systems, but also opens new avenues for research in the design of self-stabilizing protocols under severe resource limitations: constant space-complexity may not systematically prevent the existence of self-stabilizing algorithms for important non-trivial tasks.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Mathematics of computing → Discrete mathematics

Keywords and phrases Distributed algorithms, fault-tolerance, transient faults, self-stabilization, memory optimization

Digital Object Identifier 10.4230/LIPIcs.DISC.2025.17

Funding *Lélia Blin*: Additional support from ANR projects ENEDISC (ANR-24-CE48-7768-01), and from the InIDEX Project METALG.
Franck Petit: Additional support from ANR project ANR SKYDATA (ANR-22-CE25-0008-02).
Sébastien Tixeuil: Additional support from ANR project SAPPORO (2019-CE25-0005).

1 Introduction

This paper addresses the challenge of developing memory-efficient self-stabilizing algorithms for the Breadth-First Search (BFS) Spanning Tree Construction problem. Self-stabilization [3, 18, 20, 23] is a versatile technique that facilitates recovery after arbitrary *transient* faults impact the distributed system, affecting both the participating processes and the communication medium. Essentially, a self-stabilizing protocol can restore the system to a legitimate configuration (from which its behavior satisfies its specification), beginning from an arbitrary, potentially corrupted, initial global state, without the need of any human intervention. A BFS spanning tree construction involves *(i)* each node identifying a single

neighbor as its parent, with the exception of the root node, which has no parent, and *(ii)* the set of parent-child relationships forming an acyclic connected structure, ensuring that following the parent nodes from any node results in the shortest path to the root node. Memory efficiency pertains to the amount of information transmitted to neighboring nodes to enable stabilization. A smaller space complexity results in reduced information transmission, which *(i)* decreases the overhead of self-stabilization in the absence of faults or after stabilization, and *(ii)* facilitates the integration of self-stabilization and replication [22, 24].

We adopt the classical *state model* introduced in [18]. In this model, a node can read its own memory and the memory of its neighbors, and execute its algorithm accordingly when needed. This means that a node's memory is accessible to all its neighbors, ensuring they all obtain the same information. Consequently, if a node v wants to communicate information exclusively to its neighbor u, v must indicate in its memory that the information is intended only for u. This indication is typically made by adding node u's identifier. This type of communication is inherent to wireless networks, where nodes exchange information by broadcasting messages over one or more carrier waves. In such networks, the notion of *n*eighboring node of v refers to any other node able to directly receiving messages emitted by v, typically without the aid of relays. Consequently, targeting a specific neighbor necessitates that the sender explicitly includes the recipient's ID within the transmitted message. This identification requirement adds a significant overhead in memory space, specifically $O(\log n)$ bits in a network containing n nodes. Within anonymous semi-uniform networks, nodes are devoid of identifiers, and only a single node assumes a role distinct from the rest. More precisely, the distinguished node operates under a dedicated protocol, whereas all other nodes execute an identical protocol. As a result of anonymity, it is not possible to rely on identifiers to interact with or refer to a specific node.

In [21], the authors introduce the notion of a *silent algorithm* (defined in a model slightly different from the state model [18]), a fundamental concept that significantly influences the space complexity of self-stabilizing algorithms. An algorithm is said to be *silent* if every execution eventually reaches a point beyond which the state of all nodes remains unchanged. They show that in networks of n nodes, solving tasks such as leader election or spanning tree construction silently requires at least $\Omega(\log n)$ bits of memory per node. However, in the state model, a self-stabilizing algorithm is not necessary silent, and *talkative* (*i.e.*, non-silent) approaches were successful in reducing the memory cost of self-stabilizing algorithms [10, 11].

In general networks, such as those considered in this paper, self-stabilizing leader election is closely linked to self-stabilizing tree construction. On one hand, the presence of a leader enables efficient self-stabilizing tree construction [9, 13, 15, 16, 19, 25, 26, 27]. On the other hand, growing and merging trees is the primary technique for designing self-stabilizing leader election algorithms in networks, as the leader often serves as the root of an inward tree [1, 2, 4, 7, 10, 11].

From a memory perspective, the aforementioned works employ two algorithmic techniques for self-stabilization that negatively impact memory complexity. The first technique involves using a *distance* variable to store the distance of each node to the elected node in the network. This distance variable is employed in self-stabilizing spanning tree construction to break cycles resulting from arbitrary initial states, see for instance [1, 2, 4]. Clearly, storing distances in n-node networks may require $\Omega(\log n)$ bits per node. Some works [6, 10, 11] distribute pieces of information about the distances to the leader among the nodes using different mechanisms, allowing for the storage of $o(\log n)$ bits per node. However, in general networks, the best result so far [11] uses $O(\log\log n + \log\Delta)$ bits per node, closely matched by the $\Omega(\log\log n)$ bits per node lower bound obtained in [8].

The second technique involves using a *pointer-to-neighbor* variable to unambiguously designate a specific neighbor for each node (with the designated neighbor being aware of this). For tree construction, pointer-to-neighbor variables typically store the parent node in the constructed tree (with the parent being aware of its children). In a naive implementation, the parent of each node is designated by its identifier, requiring $\Omega(\log n)$ bits for each pointer variable. However, it is possible to reduce the memory requirement to $O(\log \Delta)$ bits per pointer variable in networks with a maximum degree of Δ by using node-coloring at distance 2 instead of identifiers to identify neighbors. The best available deterministic self-stabilizing distance-2 coloring protocol [11] achieves a memory footprint of $O(\log \log n + \log \Delta)$ bits per node, assuming unique identifiers. This implies that even self-stabilizing protocols solely relying on a constant number of pointer-to-neighbor variables per node [26, 16] actually use at best $O(\log \log n + \log \Delta)$ bits per node (assuming unique identifiers) in the classical state model, and that a lower bound of $\Omega(\log \Delta)$ bits per node also holds for these solutions (regardless of the nodes being anonymous or not).

We now turn to the results of Itkis and Levin [25], which constitute one of the works most closely related to ours. their approach, the authors propose a randomized asynchronous self-stabilizing algorithm for leader election, based on a *deterministic* construction of a BFS tree. Their algorithms are designed under the same model as in this paper, namely the *state model*, but augmented with *pointer-to-neighbor variables*. As already mentioned above, using such additional assumption, each node u can maintain a unique pointer to a neighbor $v \in N(u)$; more importantly, this neighbor v is aware that it is being pointed to by u. Three deterministic BFS construction algorithms are presented by Itkis and Levin [25], proving successive improvements on their space complexity. Their first algorithm is a standard BFS construction based on the distance from each node to the root, and it is requiring $O(\log n)$ bits of memory per node. The other two algorithms are presented in natural language, and their formalization as a set of guarded rules is left undefined. The former is restricted to graphs of degree 2. It is based on a sophisticated and elegant cycle-detection mechanism based on the Thue-Morse sequence. Note that although no formal proof exist in the literature, the inherent properties of the Thue-Morse sequence and finite automata theory suggest that it is impossible to implement this sequence within finite memory. Nevertheless, the authors use a Thue-Morse-based certificate to obtain a constant space complexity per node. The second of the two algorithms generalizes the first to arbitrary graphs. This algorithm maintains a forest of pointers, and guarantees acyclicity of these pointers thanks to the use of the degree-2 algorithm. The latter is applied to each path from the root of the BFS tree to its leaves, where each path is identified thanks to a depth-first search (DFS) traversal of the BFS tree. These DFS traversals are benefiting from the pointer-to-neighbors variables. To keep the space complexity constant, the DFS traversals are performed successively. This requires to keep track of which DFS traversals have already been performed. This authors argue that this can be done using constant space complexity using the pointer-to-neighbor variables allowed by their model. However, in absence of such pointers, it is not clear how to implement this sequence of DFS traversals using less than $\Omega(\log \Delta)$ bits of memory per node [15, 17].

Overall, up to this paper, there exists no deterministic self-stabilizing solution to the spanning tree construction problem that uses a constant number of bits of memory per process. Furthermore, the reuse of known techniques makes this goal unattainable.

Our contribution. We introduce the first deterministic self-stabilizing Breath-First Search Spanning Tree construction algorithm that utilizes a constant amount of memory per node in the classical state model, irrespective of network parameters such as degree, diameter, or size. Our algorithm functions in arbitrary topology networks and does not depend on any global knowledge.

In [8], it is shown that constructing a self-stabilizing spanning tree without a pre-existing leader using constant memory is impossible. Therefore, a necessary assumption for achieving constant space per node is that the anonymous network is semi-uniform, with a single node designated as the leader, serving as the root of the spanning tree, while all other nodes are anonymous and uniformly execute the same algorithm. Additionally, for synchronization purposes, our solution assumes a fully synchronous network, where all nodes operate at the same pace in a lock-step manner.

Our solution relies on a few key components.

First, each non-root node maintains a *rank* variable, which classifies neighbors as parents (if their rank is lower), siblings (if their rank is the same), or children (if their rank is higher). In the stabilized phase, the rank value corresponds to the distance to the root modulo three (hence, the BFS property of the constructed spanning tree), an idea already suggested in various non self-stabilizing settings [5, 12, 14]. The self-stabilizing property adds a significant challenge to the task, as the initial configuration may make the rank variable induce cycles or multiple roots, from which the algorithm must recover. Since identifying the single parent of each node (if it exists) cannot make use of node identifiers (as all non-root nodes are anonymous), instead we select the parent node (among the set of parents) whose port number is minimal. However, nodes do *not* explicitly communicate the specific parent they have selected, as this would necessitate $O(\log \Delta)$ bits of memory.

The second component of our algorithm uses a similar technique as the *Power Supply* introduced by Afek and Bremler [1]. In both cases, the root continuously floods the network with tokens that have a limited lifespan to circumvent one of the main issues associated with token-based approaches: the infinite circulation of tokens within a cycle. The token's lifespan is controlled by a binary counter, where each bit is stored on a distinct node. So, the tokens generated by the root decay exponentially (half of the tokens disappear at the root's neighbors, half of the remaining ones at the neighbors' downward neighbors, and so on). Tokens are used to destroy cycles and correct inconsistent situations that may appear in arbitrary initial configurations, observing that in synchronous networks, tokens passed deterministically arrive simultaneously at all nodes equidistant from the root (and thus at all neighbors with the same rank for any given node, if the ranks are correct). The exponential decay prevents tokens from looping indefinitely in the network, without relying on any global knowledge.

Overall, our algorithm uses only 6 bits of memory per node (actually, only 36 states), and has a stabilization time of $O(2^{\varepsilon})$ time units, where ε denotes the root eccentricity (unknown to the participating nodes).

2 Preliminaries

In this section, we define the underlying distributed execution model considered in this paper, and state what it means for a protocol to be self-stabilizing.

A *distributed system* is an undirected connected graph, $G = (V, E)$, where V is a set of vertices (or nodes) – $|V| = n,\ n \geq 2$ – and E is the set of edges connecting those vertices. The distance between two nodes u and v, denoted by d_v^u is the length of the shortest path

between u and v in G. Note that since G is not oriented, $d_v^u = d_u^v$, and if $u = v$, then $d_v^u = 0$. Then, the eccentricity of a node v is the maximum distance from v to any other node in the network.

Vertices represent processes, and edges represent bidirectional communication links (that is, the ability for two nodes to communicate directly). We assume that each node v is able to distinguish each incident edge with a locally assigned unique label called *port number*. Port numbers are immutable and are not coordinated in any way: the edge (u, v) may correspond to port number k at u and to port number $k' \neq k$ at v. Let $N(v)$ be the set of port numbers, also called *set of neighbors* of v. The *degree* of v denotes $|N(v)|$.

We consider the state model in the context of semi-uniform anonymous networks. In this model, the communication paradigm allows each node to read its own memory as well as the memories of its neighbors. Let v be any node in V, and let $\{u_0, \ldots, u_{|N(v)|}\}$ denote its set of neighbors. No neighbor $u_i,\ i \in \{0, \ldots, |N(v)|\}$, can determine which port number in $N(v)$ corresponds to the edge (v, u_i). Conversely, since nodes have no identifiers, v cannot designate any particular u_i to receive v's information. Hence, a pointer-to-neighbor cannot be implemented solely by reading the neighbors' variables.

A semi-uniform anonymous network is characterized by the absence of node identifiers and the presence of a unique distinguished node. This node serves as the root of the BFS tree and follows a distinct protocol, while all other nodes adhere to a uniform protocol. The protocol of a node consists of a set of variables and a set of (guarded) rules, of the following form:

$$< label >: < guard > \longrightarrow < statement >$$

Each node may write its own variables, and read its own variables and those of its neighboring nodes. The guard of a rule is a predicate on the variables of v and its neighbors. The statement of a rule of v updates one or more variables of v. A rule may be executed only if its guard evaluates to true. The rules are atomically executed, so the evaluation of a guard and the execution of the corresponding statement, if any, are done in one atomic step.

The *state* of a node is defined by the values of its variables. The *configuration* of a system is the product of the states of all nodes. Let Γ be the set of all possible configurations of the system. A distributed algorithm (or, protocol) $\mathcal{A}$ is a collection of binary transition relations, denoted by $\mapsto$, on Γ such that given two configurations γ_1 and γ_2, $\gamma_1 \mapsto \gamma_2$ by the atomic execution of guarded action of one or more nodes. A protocol $\mathcal{A}$ induces an oriented graph $\Gamma^{\mapsto} = (\Gamma, \mapsto)$, called the *transition graph* of $\mathcal{A}$. A sequence $e = \gamma_0, \gamma_1, \ldots, \gamma_i, \gamma_{i+1}, \ldots$, $\forall i \geq 0, \gamma_i \in \Gamma$, is called an *execution* of $\mathcal{A}$ if and only if $\forall i \geq 0, \gamma_i \mapsto \gamma_{i+1}$.

A node v is said to be *enabled* in a configuration $\gamma \in \Gamma$ if there exists a rule R such that the guard of R is true at v in γ. In this paper, we assume that each transition of $\Gamma^{\mapsto}$ is driven by a *synchronous scheduler*. This means that, given a pair of configurations $\gamma_1, \gamma_2 \in \Gamma$, then $\gamma_1 \mapsto \gamma_2 \in \Gamma^{\mapsto}$ if and only if all enabled nodes in γ_1 execute an atomic rule during the transition $\gamma_1 \mapsto \gamma_2$.

Self-Stabilization. A predicate P is *closed* for a transition graph $\Gamma^{\mapsto}$ if and only if every configuration in any execution e starting from a configuration satisfying P also satisfies P. A predicate Q is an *attractor* of P, denoted $P \rhd Q$, if and only if: *(i)* Q is closed for $\Gamma^{\mapsto}$, and *(ii)* for every execution e on $\Gamma^{\mapsto}$ beginning in a configuration satisfying P, e contains at least one configuration satisfying Q. A transition graph $\Gamma^{\mapsto}$ is *self-stabilizing* for a predicate P when P is an attractor of the predicate *true* (formally, *true* $\rhd P$), ensuring convergence to P from any initial configuration.

Problem to be solved. We assume a *rooted graph*, meaning that the set of vertices includes a specific node called the *root*, denoted by r. Let us call V^r the set V with the extra property that each node $v \in V$ is labeled with d_v^r. For every node $v \neq r$, let $Pred_v$ be the set of neighboring nodes of v in V^r that are labeled with $d_v^r - 1$. Let E^r be the subset of edges ($E^r \subset E$) such that for every edge $uv \in E^r$, either u belongs to $Pred_v$, or v belongs to $Pred_u$. So, in $G^r = (V^r, E^r)$, no two neighbors have the same label. Then $\overrightarrow{G^r}$ is the directed version of G^r every edge is oriented toward the smaller label. By construction, $\overrightarrow{G^r}$ is a Directed Acyclic Graph (DAG) rooted in r such that every decreasing label path from every $v \in V^r$ is a path of minimum length between v and r. Note that this DAG is uniquely defined. In the remainder of the paper, we refer to $\overrightarrow{G^r}$ as $BFS\text{-}DAG$ (rooted at r).

Let $BFS^{\equiv 3}$ be the projection of the $BFS\text{-}DAG$, where each vertex v is labeled with $d_v^r \bmod 3$. The problem we consider in this paper consists in the design of a self-stabilizing distributed algorithm that builds the $BFS^{\equiv 3}$ using a constant amount of memory. From this construction, one can retrieve the BFS tree by choosing for each node (except the root) the parent port number of minimum value.

3 Algorithm

3.1 Overview of the algorithm

Our algorithm formally described in Algorithm 1 aims to build a distributed version of $BFS^{\equiv 3}$, meaning each node v computes its distance to the root r modulo 3. Our algorithm makes use of an infinite flow of tokens originating from the root downwards the BFS-DAG. Of course, since the initial configuration is arbitrary, the nodes' variables may not originally match the $BFS^{\equiv 3}$, so the tokens may not initially flow according to the BFS-DAG.

For this algorithm to function properly, it is crucial that only the root *creates* tokens. When the root creates a token, all of its neighbors take that token in the next synchronous step, which implies that after one step, there are as many tokens as neighbors of the root. This process is then repeated from parents to children (according to the distance modulo 3 variables).

Since our model is synchronous, all tokens generated by the root at a given step progress while preserving an identical distance from the root.

A node v updates its distance modulo 3 based on the distance (modulo 3) of the neighbors from which it receives one or more tokens at the same synchronous step; if multiple neighbors send tokens, v merges them into a single token.

A common issue when using tokens to construct a spanning tree starting from an arbitrary configuration is the possibility of infinite token circulation within cycles, which prevents a valid spanning tree from forming. This problem may occur if tokens (not generated by the root) are already present in the initial configuration.

To address this, our algorithm ensures that each token has a limited lifespan, thereby avoiding infinite circulation. This limited lifespan is based on the following idea: each node maintains a bit, whose value indicate whether the token should be forwarded downwards; each time a token is received, the bit is flipped, effectively dropping half of the tokens that flow through the node. Globally, this process can be seen as creating a binary word along the paths from the root to the nodes. Tokens move along these paths, effectively performing binary addition one by one. A node v is allowed to collect a token if and only if all upward nodes having sent their tokens have their bit set to 1 before the addition. Otherwise, node v does not forward the tokens, leading to their disappearance.

Consider an execution of this approach on a chain of length ℓ, where all nodes v_1 through v_ℓ have their bits set to 0. Consequently, the binary word of length ℓ is $0000\ldots00$, note that the root does not contribute to this binary word. For simplicity, assume that a token moves from one neighbor to the next in just one step. If the root offers a token, its neighbor v_0 does not pass the token to its neighbor v_1 but changes its bit to 1. The resulting binary word is $1000\ldots00$. Next, when v_1 acquires the token, it can pass the token to v_2. At this point, v_1 changes its bit to 0 and v_2 changes its bit to 1, but the token is not forwarded to v_3 and thus disappears. The resulting binary word is $0100\ldots00$. This process continues accordingly. From this scenario, observe that the token's lifespan is very short and that for the node farthest from the root to receive the token, it may have to wait 2^ℓ steps.

This approach also circumvents one of the major issues in self-stabilizing spanning tree construction: the spanning tree is built here from the root toward the other nodes, and the root never waits for information from the spanning tree – typically expected from leaves up to the root – thus avoiding deadlock when no leaves exist (*i.e.*, when only cycles map the network).

Although our algorithm is designed so that the distance modulo 3 of each node is eventually stable (*i.e.*, eventually remains unchanged), Algorithm 1 is not silent. Indeed, even when the distributed version of $BFS^{\equiv 3}$ is built, an underlying token-passing mechanism keeps operating indefinitely.

3.2 Variables

Each node v maintains three variables. A variable that, upon stabilization of the algorithm, stores the distance to the root modulo 3, this variable, called the **rank**, is denoted by k. Consequently, for each node v, the variable k_v takes values in $\{0, 1, 2, \perp\}$. As explained in the sequel, the extra value $\perp$ is used to handle errors. When the system is stabilized, *i.e.*, Variable k_v is supposed to induce a kinship relationship matching $BFS^{\equiv 3}$. However, in the arbitrary initial configuration, this variable may be incorrect, affecting the relationship that should normally exist to form $BFS^{\equiv 3}$. Hence, cycles (induced by the parent relationships in the graph G) and incorrect DAGs (*i.e.*, DAGs rooted on nodes other than the root, or with multiple roots) may exist in an initial configuration.

Therefore, we add a token diffusion mechanism to stabilize k_v. The circulation of tokens is handled by the variable t. This variable takes values in $\{true, false, wait\}$. The states *true* and *false* indicate whether a node holds a token or not. The *wait* state serves two purposes: it prevents a node that already holds a token from immediately receiving another token, and it informs a token-holding node about which node(s) sent it the token. After stabilization, nodes cycle through the states *false*, *true*, and *wait*, in that order. A node with a *true* token moves to a *wait* token in the subsequent step, while a node with a *wait* token moves to a *false* token in the subsequent step. A node may remain in the *false* state for more than one step.

The last local variable, called **bit**, and denoted by b, is used to construct binary words originating at the root. For each node v, the variable b_v can take values in $\{\mathbf{0}, \mathbf{1}, \top\}$. The states **0** and **1** are the normal states of a bit, and as explained further, $\top$ is used to manage a reset mechanism when an error is detected locally.

Note that those three variables are also present at the root r. But, they are considered as constant with the following values: $k_r = 0$, $t_r = true$, and $b_r = \mathbf{1}$. The root has no rules to execute. Note that since the root has $t_r = true$ and $b_r = \mathbf{1}$, it *continuously* proposes a token to its neighbors.

3.3 Parent-child relationships sets and predicates

In the following, we denote by $N(v)$ the set of neighbors of node v, and by $pt_{(v,u)}$ the port number of v leading to u. The parents relationship is defined by the rank variable (*i.e.,* k) of each node. Let us denote by $P(v)$ the set of v's parents: $P(v) = \{u \in N(v) | k_u = (k_v - 1) \mod 3\}$.

To obtain a BFS spanning tree, all nodes must choose a single parent among its parents' set $P(v)$. We use the minimum port number to break such symmetric cases.

$$p(v) = \begin{cases} u | w \in P(v) \wedge pt_{(v,u)} = \min\{pt_{(v,w)}\} & \text{if } P(v) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

Children of node v, noted $C(v)$, are formally defined as: $C(v) = \{u \in N(v) | k_u = (k_v + 1) \mod 3\}$

Let $S(v)$ be the set of siblings of node v defined as follow: $S(v) = \{u \in N(v) | k_u = k_v\}$

3.4 Algorithm `TokBin`

Algorithm 1 Algorithm `TokBin` for Node $v \in V \setminus \{r\}$.

R_{er}	:	$b_v \neq \top \wedge Er(v) \longrightarrow t_v := wait; b_v := \top$
R_{reset}	:	$k_v \neq \bot \wedge b_v = \top \longrightarrow k_v := \bot; t_v := false; b_v := \mathbf{0}$
R_{erRank}	:	$\neg Er(v) \wedge k_v \neq \bot \wedge b_v \neq \top \wedge t_v = false \wedge TakeO(v) \longrightarrow t_v := wait; b_v := \top$
R_{join}	:	$\neg Er(v) \wedge k_v = \bot \wedge TakeR(v) \longrightarrow k_v := k_t(v); b_v := \mathbf{1}$
R_{tok}	:	$\neg Er(v) \wedge k_v \neq \bot \wedge t_v = false \wedge TakeP(v) \longrightarrow t_v := true$
R_{add}	:	$\neg Er(v) \wedge t_v = true \longrightarrow t_v := wait; b_v := (b_v + 1) \mod 2$
R_{ready}	:	$\neg Er(v) \wedge t_v = wait \longrightarrow t_v := false$

Our algorithm `TokBin` (see Algorithm 1) is composed of seven rules. Note that our approach is not silent: once the system has stabilized, rules $\underline{R_{tok}}$, $\underline{R_{add}}$, and $\underline{R_{ready}}$ are activated infinitely often by all nodes in the network. The others rules are executed in the stabilizing phase only.

Predicates, Macros, and Functions used in Algorithm 1 are formally defined in Subsections 3.4.1 and 3.4.2.

We use the state $\top$ of variable b_v to indicate that a node v is in an error state. Consequently, rule $\underline{R_{er}}$ detects local errors: a node v that is not in error yet observes an error (predicate $Er(v)$) activates this rule to transition to an error state. In the next step, a node already in an error state resets itself via rule $\underline{R_{reset}}$.

For our algorithm to function correctly, it is necessary to clean as much as possible of the network when an error is detected. So, when an error is detected when executing rule $\underline{R_{er}}$, the predicate $Er(v)$ of its neighbors also becomes true, allowing the the error to propagate when the neighbors also execute rule $\underline{R_{er}}$. This process may not propagate to the entire network if the root is an articulation point of the graph (that is, the graph minus the root node G^* is disconnected). However, the error does propagate to the entire connected component of G^*.

Rules $\underline{R_{tok}}$, $\underline{R_{add}}$, and $\underline{R_{ready}}$ govern token circulation. The system is synchronous, and once the algorithm converges, node ranks remain unchanged. Therefore, when tokens circulate, a node v must receive its token(s) from every parent u satisfying $b_u = 1$. These conditions are captured by the predicate $TakeP(v)$, described formally later. If $TakeP(v)$ is true, v executes rule $\underline{R_{tok}}$ to acquire a token (possibly merging its parents' tokens). When a node has a token, its children must retrieve it if its bit is **1**; otherwise, the token disappears. In either case, the node holding the token releases it by executing rule $\underline{R_{add}}$, which also increments its b variable. Note that, to enable neighbors with tokens to detect which nodes have sent one token, rule $\underline{R_{add}}$ sets the token variable to *wait* rather than directly setting it to $false$. This delay simplifies the proof of the algorithm. Finally, after sending a token and incrementing its variable b_v, node v can execute rule $\underline{R_{ready}}$ to indicate that it is ready to receive a new token. Figure 1 illustrates a possible execution of these three rules.

Observe that a node v must receive token(s) from all of its parents to maintain the parent-child relationships; otherwise, v detects an error. This error is captured by predicate $TakeO(v)$, described formally below. In such a case, v executes rule $\underline{R_{erRank}}$ to declare itself in an error state.

The last rule $\underline{R_{join}}$, concerns nodes that have been reset, and therefore do not hold a rank. A reset node v that is offered a token by a set of nodes P' – all sharing the same rank (predicate $TakeR(v)$) – joins the spanning structure by taking the nodes in P' as its parents. To do so, v adjusts its rank based on the nodes in P' and also acquires a token.

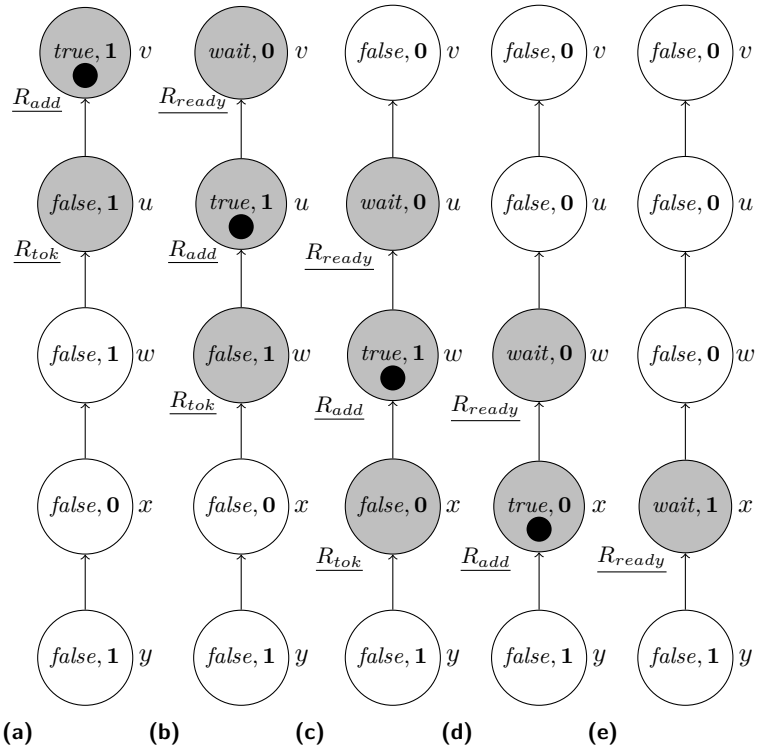

Figure 1 The gray nodes in each subfigure represent activatable nodes. (a) Node v possesses and offers a token. Node u acquires the token by applying the rule $\underline{R_{tok}}$, and node v increments its bit using $\underline{R_{add}}$. (b) Node v applies rule $\underline{R_{ready}}$ to indicate that it is ready to receive a new token. Node u holds the token, it releases the token and increments its bit by applying $\underline{R_{add}}$. Simultaneously, the node w executes rule $\underline{R_{tok}}$ to acquire the token. (d) Node x holds the token; however, node y does not accept it because node x has $b_x = \mathbf{0}$.

3.4.1 Token reception sets and predicates

In the remainder, any node satisfying predicate $Reset(v)$ is referred to as a reset node:

$$Reset(v) \equiv k_v = \perp \wedge t_v = false \wedge b_v = \mathbf{0} \tag{1}$$

In our approach, a node v may receive tokens from a neighbor u if and only if the neighbor's binary variable satisfies $b_u = \mathbf{1}$. The set $Tok(v)$ represents the neighbors of v that may send a token to v.

$$Tok(v) = \{u \in N(v) | t_u = true \wedge b_u = \mathbf{1}\} \tag{2}$$

For our purpose, a distinction is made based on which node transmits the token to v. The two following predicates apply when node v has a valid rank (*i.e.,* $k_v \neq \perp$), allowing it to identify its parent set. The predicate $TakeP(v)$ holds if all proposed tokens are from all parents of v, while predicate $TakeO(v)$ holds if at least one token sender is not the parent of v, or if not all parents of v propose a token.

$$TakeP(v) \equiv Tok(v) \neq \emptyset \wedge Tok(v) = P(v) \tag{3}$$

$$TakeO(v) \equiv Tok(v) \neq \emptyset \wedge Tok(v) \neq P(v) \tag{4}$$

The final predicate in this section is used by a reset node v. It verifies that all token sender nodes share the same rank, and that no node outside the token sender set possesses that same rank.

$$Tok_{rk}(v) = \min\{k_u | u \in Tok(v)\}$$

$$TakeR(v) \equiv Tok(v) \neq \emptyset \wedge (\forall u, w \in Tok(v) | k_u = k_w) \wedge (\forall u \in N(v) \setminus Tok(v) | k_u \neq Tok_{rk}(v)) \tag{5}$$

When a new reset node v satisfies predicate $TakeR(v)$, it can join the spanning structure by becoming a child of nodes in $Tok(v)$. In doing so, v adjust its rank accordingly, using function $k_t(v)$:

$$k_t(v) = (\{k_u | u \in Tok(v)\} + 1) \mod 3 \tag{6}$$

3.4.2 Error Detection sets and predicates

Let us introduce some predicates for all nodes in $V \setminus \{r\}$. Inconsistencies between variables can arise due to a corrupted initial configuration, necessitating detection by the algorithm. A node v with an invalid rank ($k_v = \perp$) is considered a reset node, and thus all its variables must reflect the state of a reset node. When a node v detects an error, it sets $b_v = \top$. After detecting an error, the decision is made to clean the network. Consequently, any non-reset node that has an erroneous neighbor transitions to an error state. In other words, errors propagate from a node to its neighborhood, as captured by the following predicate.

$$Er_{prg}(v) \equiv \exists u \in N(v) | k_u \neq \perp \wedge b_v = \top \tag{7}$$

Predicate $Er_{var}(v)$ detects inconsistencies between variables of a reset node.

$$Er_{var}(v) \equiv \neg Er_{prg}(v) \wedge b_v \neq \top \wedge (k_v = \perp \wedge (t_v \neq false \vee b_v \neq 0)) \tag{8}$$

If a node v has at least one child with a token, and v has not sent a token (i.e., $\neg(t_v = wait \wedge b_v = \mathbf{0})$), then v detects an error thanks to predicate $Er_{tp}(v)$.

$$Er_{tp}(v) \equiv \neg Er_{prg}(v) \wedge b_v \neq \top \wedge C(v) \neq \emptyset \wedge (\exists u \in C(v) | t_u = true) \wedge \neg(t_v = wait \wedge b_v = \mathbf{0}) \quad (9)$$

We deal with an synchronous scheduler, and we construct a $BFS^{\equiv 3}$. As a consequence, all parents of a node v must have the same state (see predicate $ErP(v)$), all children of v must share the same state (see predicate $ErC(v)$), and all siblings of v must be in the same state as v (see predicate $ErS(v)$). Moreover, a node that has a valid rank, children and no token must not have any reset nodes as neighbors (see predicate $Rt\_nd(v)$). In a normal execution of the algorithm, when a node offers the token, all nodes that do not have a valid rank join it as children. Consequently, a node cannot end up with both children and neighbors lacking a valid rank. All these constrains are captured by the following predicates.

$$ErP(v) \equiv (\exists u, w \in P(v) | b_u \neq b_w \vee t_u \neq t_w) \vee (k_v \neq \perp \wedge P(v) = \emptyset) \quad (10)$$

$$ErC(v) \equiv \exists u, w \in C(v) | b_u \neq b_w \vee t_u \neq t_w \quad (11)$$

$$ErS(v) \equiv \exists u \in S(v) | b_u \neq b_v \vee t_u \neq t_v \quad (12)$$

$$Rt\_nd(v) \equiv k_v \neq \perp \wedge C(v) \neq \emptyset \wedge t_v \neq true \wedge (\exists u \in N(v) \mid k_u = \perp) \quad (13)$$

$$Er_N(v) \equiv \neg Er_{prg}(v) \wedge b_v \neq \top \wedge \neg TakeO(v) \wedge \big(ErP(v) \vee ErC(v) \vee ErS(v) \vee Rt\_nd(v)\big) \quad (14)$$

A node v with a valid rank ($k_v \neq \perp$) must have a non empty set of parents, otherwise the node is in error:

$$FalseR(v) \equiv k_v \neq \perp \wedge P(v) = \emptyset \quad (15)$$

Neighbors u and w of a reset node v with the same rank must have identical values for variables b and t. Otherwise, v detects an error and enters the error state (i.e., $b_v = \top$) so that, in the next step, nodes u and w can also transition to the error state. Predicate $Er_\pi(v)$ is dedicated to this purpose:

$$Er_\pi(v) \equiv Reset(v) \wedge (\forall u, w \in N(v) | k_u = k_w \wedge (b_u \neq b_w \vee t_u \neq t_w)) \quad (16)$$

A node v can then apply the rest of the rules of the algorithm if it does not find an error in its neighborhood, and it is not in an error state itself. Predicate $Er(v)$ captures this.

$$Er(v) \equiv Er_{var}(v) \vee Er_{tp}(v) \vee Er_N(v) \vee FalseR(v) \vee Er_{prg}(v) \vee Er_\pi(v) \quad (17)$$

4 Main Result

We now state our main technical result.

▶ **Theorem 1.** *In a semi-uniform model where r is the distinguished node, and every other node is anonymous, our synchronous deterministic self-stabilizing algorithm* `TokBin` *constructs a BFS tree rooted in r using $O(1)$ bits per node in $O(2^\varepsilon)$ steps, where ε denotes the eccentricity of r.*

Overview of Correctness. To establish space complexity, we simply observe that each node $v \in V$ maintains three local variables: k_v, t_v, and b_v. Together, those variables require only 6 bits, ensuring a per-node space complexity of $O(1)$ bits. The correctness proof is carried out in several stages. First, we demonstrate – using a potential function – that starting from an arbitrary configuration, our system converges and remains within a subset of configurations denoted by $\Gamma_{\bar{e}}$. More precisely, $\Gamma_{\bar{e}}$ is the set of configurations where obvious errors $Er_{var}(\gamma_0, v) = true$, $Er_{tp}(\gamma_0, v) = true$, or $Er_N(\gamma_0, v) = true$ (see Equations 8, 9, and 14) are no longer present.

Afterwards, we define the set of legal configurations Γ^*. A configuration is considered legal if, for every node v, v's rank is equal to its distance from the root in the graph modulo 3. Moreover, we must ensure that the nodes' ranks no longer change. In our synchronous setting, this requires that all shortest paths from the root to any node v are identical with respect to the nodes' states on the paths. In other words, every node v at the same distance from the root must be in the same state (*i.e.*, has identical values for variables t_v and b_v).

Then, we need to prove that, *(i)* starting from a possibly illegal (but free of obvious errors) configuration in $\Gamma_{\bar{e}}$, the system converges to a legal configuration, and *(ii)* the system remains in the set of legal configurations Γ^*, thanks to our algorithm. Note that in $\Gamma_{\bar{e}}$, only the rules $\underline{R_{tok}}$, $\underline{R_{add}}$, and $\underline{R_{ready}}$ are executable – these are the rules that govern token circulation.

The second part of the proof (closure) is established by observing that at each step of the algorithm execution, the predicate defining legal configurations is maintained. The first par of the proof (convergence) is more involved, and is broken into several steps.

The remainder of the correctness proof is as follows. First, we show that within $O(n)$ steps, all tokens present in the initial configuration are destroyed, resulting in configurations belonging to the set Γ_{tr}. A key component in achieving this is the predicate $Er_{tp}(v)$ (see Equation 9) and the rule $\underline{R_{add}}$, which modifies the variable b. Recall that a node can receive the token if and only if its parent p has $b_p = \mathbf{1}$. Therefore, if a node v receives the token when its parent offers it (*i.e.*, when $b_p = \mathbf{1}$), then the next time the parent offers it, v cannot accept the token since the parent has $b_p = 0$.

Now we can prove that, starting from a configuration in Γ_{tr}, if a node receives the token, it must have a legal rank (*i.e.*, $k_v = d_v^r \mod 3$). Next, we formalize the paths originating from the root to demonstrate that any node v whose every path from the root is legal receives the token in at most $O(2^{d_v^r - 1})$ steps, where d_v^r is the distance from the root r to v in the graph (d_v^r is bounded by ε).

With these results, we have all the necessary tools to prove that our system converges to a set of error-free configurations, denoted by Γ_{cl}. More precisely, Γ_{cl} comprises configurations where nodes are either legal or have been restarted. Finally, starting from a configuration in Γ_{cl}, we prove convergence toward the set of legal configurations Γ^*.

5 Concluding Remarks

We presented the first constant space deterministic self-stabilizing BFS tree construction algorithm that operate in semi-uniform synchronous networks of arbitrary topology. Our solution `TokBin` is independent of any global network parameter (maximum degree, diameter, number of nodes, etc.). As classical techniques such as point-to-neighbor and distance-to-the-root cannot be used in such a constrained setting, we introduced a novel infinite token stream technique to break cycles and restore BFS construction that may be of independent interest for other tasks. We briefly mention how our BFS construction can be used to optimally color

bipartite graphs (that is, with two colors BLACK and WHITE). All nodes run `TokBin` so that eventually a $BFS^{\equiv 3}$ is constructed. With respect to colors, the root has color BLACK and retains it forever, while every other node take the opposite color of its parents whenever it executes `TokBin` (if it has parents). After stabilization of the $BFS^{\equiv 3}$ construction, all children of the root take the same color, and after one step their children take the root color, and so on, so that a 2-coloring of the bipartite graph is eventually achieved. This very simple application of `TokBin` still uses constant memory, as opposed to dedicated previous solutions to the same problem found in the literature [29, 28] that require $\Omega(\log n)$ bits per node.

Two important open problems remain. First, our solution heavily relies on the network operation being fully synchronous (all nodes execute code in a lock-step synchronous fashion). It would be interesting to extend our token stream technique to an asynchronous setting. In this setting, we expect that some synchronization technique will be necessary, and it remains unclear whether such additional mechanism can be done in constant space. Second, while our algorithm uses a constant amount of memory per node, its time complexity (measured in steps) is exponential. This is due to the exponential decay mechanism for tokens that may require the root to send an exponential number of them to finally reach the furthest nodes in the network. One could speed up this process by adding more states to the b variables: if the domain of b becomes $\{\mathbf{0}, \mathbf{1}, \ldots, \mathbf{k}, \top\}$, and token are accepted for any value that is not $\mathbf{0}$ or $\top$ (assuming b is incremented modulo k anytime the incoming token is accepted), less tokens are destroyed, but the overall time complexity remains exponential. By contrast, solutions that make use of $O(\log \Delta + \log \log n)$ bits per node have polynomial time complexity [10]. Whether there exists a constant space and polynomial time solution to the problem is left for future research.

References

1 Yehuda Afek and Anat Bremler-Barr. Self-stabilizing unidirectional network algorithms by power supply. *Chic. J. Theor. Comput. Sci.*, 1998, 1998. URL: `http://cjtcs.cs.uchicago.edu/articles/1998/3/contents.html`.

2 Yehuda Afek, Shay Kutten, and Moti Yung. Memory-efficient self stabilizing protocols for general networks. In Jan van Leeuwen and Nicola Santoro, editors, *Distributed Algorithms, 4th International Workshop, WDAG '90, Bari, Italy, September 24-26, 1990, Proceedings*, volume 486 of *Lecture Notes in Computer Science*, pages 15–28. Springer, 1990. `doi:10.1007/3-540-54099-7_2`.

3 Karine Altisen, Stéphane Devismes, Swan Dubois, and Franck Petit, editors. *Introduction to Distributed Self-Stabilizing Algorithms*. Synthesis Lectures on Distributed Computing. Morgan & Claypool Publishers, 2019.

4 Anish Arora and Mohamed G. Gouda. Distributed reset. *IEEE Trans. Computers*, 43(9):1026–1038, 1994. `doi:10.1109/12.312126`.

5 James Aspnes. Distributed breadth first search, 2005. Lecture notes of Distributed Computing, Fall 2005, http://pine.cs.yale.edu/pinewiki/DistributedBreadthFirstSearch.

6 Baruch Awerbuch and Rafail Ostrovsky. Memory-efficient and self-stabilizing network {RESET} (extended abstract). In James H. Anderson, David Peleg, and Elizabeth Borowsky, editors, *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing, Los Angeles, California, USA, August 14-17, 1994*, pages 254–263. ACM, 1994. `doi:10.1145/197917.198104`.

7 Lélia Blin, Shlomi Dolev, Maria Gradinariu Potop-Butucaru, and Stephane Rovedakis. Fast self-stabilizing minimum spanning tree construction - using compact nearest common ancestor labeling scheme. In Nancy A. Lynch and Alexander A. Shvartsman, editors, *Distributed Computing, 24th International Symposium, DISC 2010, Cambridge, MA, USA, September*

13-15, 2010. Proceedings, volume 6343 of *Lecture Notes in Computer Science*, pages 480–494. Springer, 2010. doi:10.1007/978-3-642-15763-9_46.

8 Lélia Blin, Laurent Feuilloley, and Gabriel Le Bouder. Optimal space lower bound for deterministic self-stabilizing leader election algorithms. *Discret. Math. Theor. Comput. Sci.*, 25(1), 2023. doi:10.46298/DMTCS.9335.

9 Lélia Blin, Maria Potop-Butucaru, and Stephane Rovedakis. A super-stabilizing log(n)log(n)-approximation algorithm for dynamic steiner trees. *Theor. Comput. Sci.*, 500:90–112, 2013. doi:10.1016/J.TCS.2013.07.003.

10 Lélia Blin and Sébastien Tixeuil. Compact deterministic self-stabilizing leader election on a ring: the exponential advantage of being talkative. *Distributed Computing*, 31(2):139–166, 2018. doi:10.1007/s00446-017-0294-2.

11 Lélia Blin and Sébastien Tixeuil. Compact self-stabilizing leader election for general networks. *J. Parallel Distributed Comput.*, 144:278–294, 2020. doi:10.1016/J.JPDC.2020.05.019.

12 Christian Boulinier, Ajoy Kumar Datta, Lawrence L. Larmore, and Franck Petit. Time and space optimal distributed BFS tree construction. *Information Processing Letters*, 108(5):273–278, 2008. doi:10.1016/j.ipl.2008.05.016.

13 NS Chen, HP Yu, and ST Huang. A self-stabilizing algorithm for constructing spanning trees. *Information Processing Letters*, 39:147–151, 1991. doi:10.1016/0020-0190(91)90111-T.

14 Reuven Cohen, Pierre Fraigniaud, David Ilcinkas, Amos Korman, and David Peleg. Label-guided graph exploration by a finite automaton. *ACM Trans. Algorithms*, 4(4):42:1–42:18, 2008. doi:10.1145/1383369.1383373.

15 Zeev Collin and Shlomi Dolev. Self-stabilizing depth-first search. *Inf. Process. Lett.*, 49(6):297–301, 1994. doi:10.1016/0020-0190(94)90103-1.

16 Ajoy K. Datta, Stéphane Devismes, Colette Johnen, and Lawrence L. Larmore. Analysis of a memory-efficient self-stabilizing BFS spanning tree construction. *Theor. Comput. Sci.*, 955:113804, 2023. doi:10.1016/J.TCS.2023.113804.

17 Ajoy Kumar Datta, Colette Johnen, Franck Petit, and Vincent Villain. Self-stabilizing depth-first token circulation in arbitrary rooted networks. *Distributed Comput.*, 13(4):207–218, 2000. doi:10.1007/PL00008919.

18 Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974. doi:10.1145/361179.361202.

19 S Dolev, A Israeli, and S Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7:3–16, 1993. doi:10.1007/BF02278851.

20 Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000.

21 Shlomi Dolev, Mohamed G. Gouda, and Marco Schneider. Memory requirements for silent stabilization. *Acta Informatica*, 36(6):447–462, 1999. doi:10.1007/S002360050180.

22 Mohamed G. Gouda, Jorge Arturo Cobb, and Chin-Tser Huang. Fault masking in tri-redundant systems. In Ajoy Kumar Datta and Maria Gradinariu, editors, *Stabilization, Safety, and Security of Distributed Systems, 8th International Symposium, SSS 2006, Dallas, TX, USA, November 17-19, 2006, Proceedings*, volume 4280 of *Lecture Notes in Computer Science*, pages 304–313. Springer, 2006. doi:10.1007/978-3-540-49823-0_21.

23 Ted Herman. Origin of self-stabilization. In Krzysztof R. Apt and Tony Hoare, editors, *Edsger Wybe Dijkstra: His Life, Work, and Legacy*, volume 45 of *ACM Books*, pages 81–104. ACM / Morgan & Claypool, 2022. doi:10.1145/3544585.3544592.

24 Ted Herman and Sriram V. Pemmaraju. Error-detecting codes and fault-containing self-stabilization. *Inf. Process. Lett.*, 73(1-2):41–46, 2000. doi:10.1016/S0020-0190(99)00164-7.

25 Gene Itkis and Leonid A. Levin. Fast and lean self-stabilizing asynchronous protocols. In *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*, pages 226–239. IEEE Computer Society, 1994. doi:10.1109/SFCS.1994.365691.

26 C Johnen. Memory-efficient self-stabilizing algorithm to construct BFS spanning trees. In *Proceedings of the Third Workshop on Self-Stabilizing Systems*, pages 125–140. Carleton University Press, 1997.

27 Amos Korman, Shay Kutten, and Toshimitsu Masuzawa. Fast and compact self stabilizing verification, computation, and fault detection of an MST. In Cyril Gavoille and Pierre Fraigniaud, editors, *Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing, PODC 2011, San Jose, CA, USA, June 6-8, 2011*, pages 311–320. ACM, 2011. doi:10.1145/1993806.1993866.

28 Adrian Kosowski and Lukasz Kuszner. Self-stabilizing algorithms for graph coloring with improved performance guarantees. In Leszek Rutkowski, Ryszard Tadeusiewicz, Lotfi A. Zadeh, and Jacek M. Zurada, editors, *Artificial Intelligence and Soft Computing - ICAISC 2006, 8th International Conference, Zakopane, Poland, June 25-29, 2006, Proceedings*, volume 4029 of *Lecture Notes in Computer Science*, pages 1150–1159. Springer, 2006. doi:10.1007/11785231_120.

29 Sumit Sur and Pradip K. Srimani. A self-stabilizing algorithm for coloring bipartite graphs. *Inf. Sci.*, 69(3):219–227, 1993. doi:10.1016/0020-0255(93)90121-2.