Compact self-stabilizing leader election for general networks

Lélia Blin, Sébastien Tixeuil

Version of Record: https://www.sciencedirect.com/science/article/pii/S0743731520302987
Manuscript_0f7a1b60bbde09f2349620de66e626f0

Compact Self-Stabilizing Leader Election for General Networks*

Lélia Blin†
Sorbonne Université, CNRS,
Université d'Evry-Val-d'Essonne,
LIP6 UMR 7606, 4 place Jussieu 75005, Paris.
lelia.blin@lip6.fr

Sébastien Tixeuil
Sorbonne Université, CNRS
CNRS, LIP6 UMR 7606, 4 place Jussieu 75005, Paris.
sebastien.tixeuil@lip6.fr

February 6, 2020

Abstract

We present a self-stabilizing leader election algorithm for general networks, with space-complexity $O(\log \Delta + \log\log n)$ bits per node in n-node networks with maximum degree Δ. This space complexity is sub-logarithmic in n as long as $\Delta = n^{o(1)}$. The best space-complexity known so far for general networks was $O(\log n)$ bits per node, and algorithms with sub-logarithmic space-complexities were known for the ring only. To our knowledge, our algorithm is the first algorithm for self-stabilizing leader election to break the $\Omega(\log n)$ bound for silent algorithms in general networks. Breaking this bound was obtained via the design of a (non-silent) self-stabilizing algorithm using sophisticated tools such as solving the distance-2 coloring problem in a silent self-stabilizing manner, with space-complexity $O(\log \Delta + \log\log n)$ bits per node. Solving this latter coloring problem allows us to implement a sub-logarithmic encoding of spanning trees — storing the IDs of the neighbors requires $\Omega(\log n)$ bits per node, while we encode spanning trees using $O(\log \Delta + \log\log n)$ bits per node. Moreover, we show how to construct such compactly encoded spanning trees without relying on variables encoding distances or number of nodes, as these two types of variables would also require $\Omega(\log n)$ bits per node.

1 Introduction

1.1 Motivation

This paper tackles the problem of designing memory efficient self-stabilizing algorithms for the leader election problem. *Self-stabilization* [22] is a general paradigm to provide recovery capabilities to networks. Intuitively, a protocol is self-stabilizing if it can recover from any transient failure, without external intervention. *Leader election* is one of the fundamental building blocks of distributed computing, as it enables a single node in the network to be distinguished, and thus to perform specific actions. Leader election is especially important in the context of self-stabilization as many protocols for various problems assume that a single leader exists in the network, even after faults occur. Hence, a self-stabilizing leader election mechanism enables such protocols to be run in networks where no leader is given a priori, by using simple stabilization-preserving composition techniques [22].

Memory efficiency relates to the amount of information to be sent to neighboring nodes for enabling stabilization. As a result, only mutable memory (used to store variables) is considered

*A preliminary version of this paper has appeared in [13, 15].
†Additional support from the ANR project ESTATE.

for computing memory complexity of a self-stabilizing protocols, while immutable memory (used to store the code of the protocol) is not considered. A small space-complexity induces a smaller amount of information transmission, which (1) reduces the overhead of self-stabilization when there are no faults, or after stabilization [1], and (2) facilitates mixing self-stabilization and replication [31].

1.2 Related work

A foundational result regarding space-complexity in the context of self-stabilizing silent algorithms[1] is due to Dolev et al. [23], stating that in n-node networks, $\Omega(\log n)$ bits of memory per node are required for solving tasks such as leader election. So, only *talkative* algorithms may have $o(\log n)$-bit space-complexity for self-stabilizing Leader Election solution. Several attempts to design compact self-stabilizing leader election algorithms (i.e., algorithms with space-complexity $o(\log n)$ bits) were performed but restricted to rings. The algorithms by Mayer et al. [37], by Itkis and Levin [34], and by Awerbuch and Ostrovsky [7] use a constant number of bits per node, but they only guarantee *probabilistic* self-stabilization (in the Las Vegas sense). *Deterministic* self-stabilizing leader election algorithms for rings were first proposed by Itkis et al. [35] for rings with a prime number of nodes. Beauquier et al. [8] consider rings of arbitrary size, but assume that node identifiers in n-node rings are bounded from above by $n + k$, where k is a small constant. The best result known so far in this context [14] is a deterministic self-stabilizing leader election algorithm for rings of arbitrary size using identifiers of arbitrary polynomially bounded values, with space complexity $O(\log \log n)$ bits per node.

In general networks, self-stabilizing leader election is tightly connected to self-stabilizing tree-construction. On the one hand, the existence of a leader enables time- and memory-efficient self-stabilizing tree-construction [16, 24, 18, 12, 36]. On the other hand, growing and merging trees is the main technique for designing self-stabilizing leader election algorithms in networks, as the leader is often the root of an inward tree [3, 4, 2]. To the best of our knowledge, all algorithms that do not assume a pre-existing leader [3, 4, 2, 10] for tree-construction use $\Omega(\log n)$ bits per node. This high space-complexity is due to the implementation of two main techniques, used by all algorithms, and recalled below.

The first main technique is the use of a *pointers-to-neighbors* variable, that is meant to designate unambiguously one particular neighbor of every node. For the purpose of tree-construction, pointers-to-neighbors variables are typically used to store the parent node in the constructed tree. Specifically, the parent of every node is designated unambiguously by its identifier, requiring $\Omega(\log n)$ bits for each pointer variable. In principle, it would be possible to reduce the memory to $O(\log \Delta)$ bits per pointer variable in networks with maximum degree Δ, by using node-coloring at distance 2 instead of identifiers to identify neighbors. However, this, in turn, would require the availability of a self-stabilizing distance-2 node-coloring algorithm that uses $o(\log n)$ bits per node. Previous self-stabilizing distance-2 coloring algorithms use variables of large size. For instance, in the algorithm by Herman et al. [32], every node communicates its distance-3 neighborhood to all its neighbors, which yields a space-complexity of $O(\Delta^3 \log n)$ bits. Johnen et al. [30] draw random colors in the range $[0, n^2]$, which yields a space-complexity of $O(\log n)$ bits. Finally, while the deterministic algorithm of Blair et al. [9] reduces the space-complexity to $O(\log \Delta)$ bits per node, this is achieved by ignoring the cost of storing another pointer-to-neighbor variable at each node. In absence of a distance-2 coloring (which their algorithm [9] is precisely supposed to produce), their implementation still requires $\Omega(\log n)$ bits per node. To date, no self-stabilizing algorithm implement pointer-to-neighbor variables with

[1] An algorithm is *silent* if each of its executions reaches a point in time after which the states of nodes do not change. A non-silent algorithm is said to be *talkative* (see [13]).

space-complexity $o(\log n)$ bits in arbitrary networks.

The second main technique for tree-construction or leader election is the use of a *distance* variable that is meant to store the distance of every node to the elected node in the network. Such distance variable is used in self-stabilizing spanning tree-construction for breaking cycles resulting from arbitrary initial state (see [3, 4, 2]). Clearly, storing distances in n-node networks may require $\Omega(\log n)$ bits per node. There are a few self-stabilizing tree-construction algorithms that are not using explicit distance variables (see, e.g., [20]), but their space-complexity is huge (e.g. $O(n \log n)$ bits of memory per node [20]). Using the general principle of distance variables with space-complexity below $\Theta(\log n)$ bits was attempted by Awerbuch et al. [7], and Blin et al. [13]. These papers distribute pieces of information about the distances to the leader among the nodes according to different mechanisms, enabling to store $o(\log n)$ bits per node. However, these sophisticated mechanisms have only been demonstrated in rings. To date, no self-stabilizing algorithms implement distance variables with space-complexity $o(\log n)$ bits in arbitrary networks.

Our results

In this paper, we design and analyze a self-stabilizing leader election algorithm with space-complexity $O(\log \Delta + \log \log n)$ bits in n-node networks with maximum degree Δ. This algorithm is the first self-stabilizing leader election algorithm for arbitrary networks with space-complexity $o(\log n)$ (whenever $\Delta = n^{o(1)}$). It is designed for the standard state model (a.k.a. shared memory model) for self-stabilizing algorithms in networks, and it performs against the unfair distributed scheduler.

The design of our algorithm requires overcoming several bottlenecks, including the difficulties of manipulating pointers-to-neighbors, and distance variables using $o(\log n)$ bits in arbitrary networks. Overcoming these bottlenecks was achieved thanks to the development of sub-routine algorithms, each deserving independent special interest described hereafter.

First, we extend the bit-wise ring publication technique [13] to arbitrary topologies. Our approach retains the $O(\log \log n)$ bits per node complexity for storing identifiers, and is thus independent of the degree.

Second, we propose the first *silent* self-stabilizing algorithm for distance-2 coloring that breaks the space-complexity of $\Omega(\log n)$ bits of memory per nodes. More precisely this new algorithm achieves a space-complexity of $O(\log \Delta + \log \log n)$ bits of memory per nodes. As opposed to previous distance-2 coloring algorithms, we do not use full identifiers for encoding pointer-to-neighbor variables (this would require $O(\log n)$ bits per node). Instead, our compact representation of the identifiers (using $O(\log \log n)$ bits per node) enables symmetry breaking. This distance-2 coloring permits to distinguish parent and children and a node's neighbors, allowing the design of a compact encoding of spanning trees.

Third, we design a new technique to detect the presence of cycles given by the current set of pointers-to neighbors. This approach does not use distances, but it is based on the uniqueness of each identifier in the network. Notably, this technique can be implemented by a *silent* self-stabilizing algorithm, with space-complexity $O(\log \Delta + \log \log n)$ bits of memory per nodes.

Last but not least, we design a new technique to avoid the creation of cycles during the execution of the leader election algorithm. Again, this technique does not use distances but maintains a spanning forest, which eventually reduces to a single spanning tree rooted at the leader at the completion of the leader election algorithm. Implementing this technique results in a self-stabilizing algorithm with space complexity $O(\log \Delta + \log \log n)$ bits of memory per nodes.

2 Model and definitions

2.1 Protocol syntax and semantics

We consider a distributed system consisting of n processes that form an arbitrary communication graph. The processes are represented by the nodes of this graph, and the edges represent pairs of processes that can communicate directly with each other. Such processes are said to be *neighbors*. Let $G = (V, E)$ be an n-node graph, where V is the set of nodes, and E the set of edges and Δ the degree of the graph.

Space-complexity in self-stabilization considers only volatile memory (that is, memory whose content changes during the execution of the protocol), while non-volatile memory (whose content does *not* change during the execution of the protocol, used *e.g.*, to store the code and the constants, and in particular the node unique identifier) is not included in the space complexity. Volatile memory includes the space allocated for protocol variables, and in particular the program counter (that commands the next line of code to execute). So, to achieve $o(\log n)$ bits of memory per node, we define helping functions that enable volatile memory to remain below that threshold.

A node v has access to a constant unique identifier ID_v, but can only access its identifier one bit at a time, using the $\texttt{Bit}_v(i)$ function, which returns the position of the i^{th} most significant bit equal to 1 in ID_v. Even though identifiers require $\Omega(\log n)$ bits of memory per node in the worst case, the `Bit` function can be stored in the immutable code portion of the node. We present here the pseudocode for the $\texttt{Bit}_v$ function at a particular node v. Note that since nodes have unique identifiers, they are allowed to execute unique code. For example, suppose node v has identifier 10 (in decimal notation), or 1010 (in binary notation). Then, one can implement $\texttt{Bit}_v(i)$ as follows for $v = 1010$:

$$\texttt{Bit}_v(i) := \begin{cases} 4 & \text{if} \quad \text{i=1} \\ 2 & \text{if} \quad \text{i=2} \\ \text{-1} & \text{if} \quad i > 2 \end{cases}$$

Since we assume that all identifiers are $O(\log n)$ bits long, the $\texttt{Bit}_v$ function only returns values with $O(\log \log n)$ bits. Also, when executing Function $\texttt{Bit}_v$, the program counter only requires $O(\log \log n)$ values. In turn, this position can be encoded with $O(\log \log n)$ bits when identifiers are encoded using $O(\log n)$ bits, as we assume they are. A node v has access to locally unique port numbers associated with its adjacent edges. We do not assume any consistency between port numbers of a given edge. In short, port numbers are constant throughout the execution but initialized by an adversary. Each process contains variables and rules. Variable ranges over a domain of values. The variable var_v denote the variable var located at node v. A rule is of the form $\langle label \rangle : \langle guard \rangle \longrightarrow \langle command \rangle$ [21]. A *guard* is a boolean predicate over process variables. A *command* is a set of variable-assignments. A command of process p can only update its own variables. On the other hand, p can read the variables of its neighbors. This classical communication model is called the *state model* or the *state-sharing communication model*. This model is also used in stabilization-preserving compilers that produce actual code [6, 17, 19, 38].

An assignment of values to all variables in the system is called a *configuration*. A rule whose guard is **true** in some system configuration is said to be *enabled* in this configuration. The rule is *disabled* otherwise. The atomic execution of a subset of enabled rules (at most one rule per process) results in a transition of the system from one configuration to another. This transition is called a *step*. A *run* of a distributed system is a maximal alternating sequence of configurations and steps. Maximality means that the execution is either infinite or its final configuration has no rule enabled.

2.2 Schedulers

The asynchronism of the system is modeled by an adversary (a.k.a. *scheduler*) that chooses, at each step, the subset of enabled processes that are allowed to execute one of their rules during this step. Those schedulers can be classified according to their characteristics (like fairness, distribution, ...), and a taxonomy was presented By Dubois *et al.* [25]. Note that we assume here an unfair distributed scheduler. This scheduler is the most challenging since no assumption is made of the subset of enabled processes chosen by the scheduler at each step. We only require this set to be nonempty if the set of enabled processes is not empty in order to guarantee progress of the algorithm.

2.3 Predicates and specifications

A predicate is a boolean function over configurations. A configuration *conforms* to some predicate R, if R evaluates to **true** in this configuration. The configuration *violates* the predicate otherwise. Predicate R is *closed* in a certain protocol P, if every configuration of a run of P conforms to R, provided that the protocol starts from a configuration conforming to R. Note that if a protocol configuration conforms to R, and the configuration resulting from the execution of any step of P also conforms to R, then R is closed in P.

A *specification* for a processor p defines a set of configuration sequences. These sequences are formed by variables of some subset of processors in the system. This subset always includes p itself.

Problem specification prescribes the protocol behavior. The output of the protocol is carried through external variables, that are updated by the protocol, and used to display the results of the protocol computation. The problem specification is the set of sequences of configurations of external variables.

A protocol implements the specification. Part of the implementation is the mapping from the protocol configurations to the specification configurations. This mapping does not have to be one-to-one. However, we only consider unambiguous protocols where each protocol configuration maps to only one specification configuration. Once the mapping between protocol and specification configurations is established, the protocol runs are mapped to specification sequences as follows. Each protocol configuration is mapped to the corresponding specification configuration. Then, stuttering, the consequent identical specification configurations, is eliminated. Overall, a run of the protocol satisfies the specification if its mapping belongs to the specification. Protocol P *solves* problem S under a certain scheduler if every run of P produced by that scheduler satisfies the specifications defined by S. A predicate I is an *invariant* of protocol P if every run of P that starts in a state conforming to I satisfies I in every subsequent configuration. Given two predicates l_1 and l_2 for protocol P, l_2 is an *attractor* for l_1 if every run of protocol P that starts from a configuration that conforms to l_1 contains a configuration that conforms to l_2. Such a relationship is denoted by $l_1 \rhd l_2$. Also, the $\rhd$ relation is transitive: if l_1, l_2, and l_3 are predicates for P, and $l_1 \rhd l_2$ and $l_2 \rhd l_3$, then $l_1 \rhd l_3$. In this last case, l_2 is called an *intermediate* attractor towards l_3.

Definition 1 (Self-stabilization)**.** *A protocol P is self-stabilizing [22] to specification S if there exists a predicate L for P such that:*

1. *L is an attractor for true,*
2. *Any run of P starting from a configuration satisfying L satisfies S.*

Definition 2 (Leader Election)**.** *Consider a system of processes where each process' set of variables is mapped to a boolean specification variable leader denoted by ℓ. The leader election*

specification sequence consists in a single specification configuration where a unique process p maps to $\ell_p = true$, and every other process $q \neq p$ maps to $\ell_q = false$.

Definition 3. *In the state model, a protocol is silent if and only if every execution is finite. Otherwise, the protocol is talkative.*

We measure time complexity with respect to individual steps performed by each process. So, the time complexity of a self-stabilizing leader election algorithm is the highest number of individual steps before a single leader is elected, starting from an arbitrary configuration.

3 Compact self-stabilizing leader election for networks

Our new self-stabilizing leader election algorithm is based on a spanning tree-construction rooted at a maximum degree node, without using distances. If multiple maximum degree nodes are present in the network, we break ties with colors and if necessary with identifiers.

Theorem 1. *Algorithm called* **C-LE** *solves the leader election problem in a talkative self-stabilizing manner in any n-node graph, assuming the state model and a distributed unfair scheduler, with $O(\log \Delta + \log \log n)$ bits of memory per node.*

Our talkative self-stabilizing algorithm reuses and extends a technique for obtaining compact identifiers of size $O(\log \log n)$ bits of memory per node presented in Section 3.1. Then, the leader election process consists in running several algorithms layers using decreasing priorities (see also Figure 1):

1. An original silent self-stabilizing distance-2 coloring presented in subsection 3.2 that permits to implement pointer-to-neighbors with $o(\log n)$ bits of memory per node.

2. A silent self-stabilizing cycle destruction and illegitimate sub spanning tree-destruction reused from previous work [11, 13] presented in subsection 3.3.

3. A new silent self-stabilizing cycle detection that does not use distance to the root variables presented in subsection 3.4.

4. An original talkative self-stabilizing spanning tree-construction, that still does not use distance to the root variables, presented in subsection 4. This algorithm is trivially modified to obtain a leader election algorithm.

5. In section 5, we describe how to integrate all previous components into a leader election protocol for general graphs.

3.1 Compact memory using identifiers

As many deterministic self-stabilizing leader election algorithms, our approach ends up comparing node unique identifiers. However, to avoid communicating the full $\Omega(\log n)$ bits to each neighbor at any given time, we reuse the scheme devised in previous work [13] to progressively publish node identifiers. Let ID_v be the identifier of node v. We assume that $\text{ID}_v = \sum_{i=0}^{k} b_i 2^i$. Let $I_v = \{i \in \{0, ..., k\}, b_i \neq 0\}$ be the set of all non-zero bit-positions in the binary representation of ID_v. Then, I_v can be written as $\{pos_1, ..., pos_j\}$, where $pos_k > pos_{k+1}$. In the process of comparing node unique identifiers during the leader election algorithm execution, the nodes must first agree on the same bit-position pos_{j-i+1} (for $i = 1, \ldots, j$); this step of the algorithm

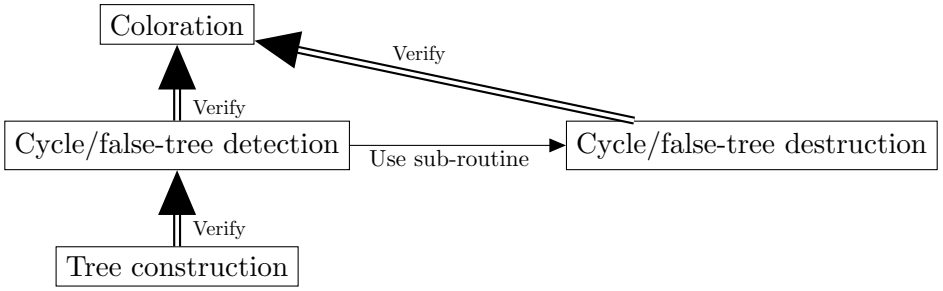

Figure 1: Overview of algorithm

defines *phase* i. Put differently, the bit-positions are communicated in decreasing order of significance in the encoding of the identifier. In turn, this may propagate it to their neighbors, and possibly to the whole network in subsequent phases. This propagation is used in the following to break symmetries in the coloring problem or to detect a cycle in spanning tree construction.

If all identifiers are in $[1, n^c]$, for some constant $c \geqslant 1$, then the communicated bit-positions are less than or equal to $c\lceil \log n \rceil$, and thus can be represented with $O(\log \log n)$ bits. However, the number of bits used to encode identifiers may be different for two given nodes, so there is no common upper bound for the size of identifiers. We circumvent this problem using a ranking on bit-positions that is agnostic on the size of the identifiers. We extract of our previous works the part dedicated to the propagation of the identifier bit by bit in phases, remark that we slightly modify our previous work. Since we do not assume that the identifiers of every node are encoded using the same number of bits, simply comparing the i-th most significant bit of two nodes is irrelevant. Instead, we use variable $\widehat{\mathsf{B}}_v$, which represents the most significant bit-position of node v. In other words, $\widehat{\mathsf{B}}_v$ represents the size of the binary representation of ID_v. The variables ph, Bp are the core of the identifier comparison process. Variable ph_v stores the current phase number i, while variable Bp_v stores the bit-position of ID_v at phase i. Remark that the number of non-zero bits can be smaller than the size of the binary representation of the identifier of the node, so if there are no more non-zero bit at phase $i \leqslant \widehat{\mathsf{B}}_v$, we use $\mathsf{Bp}_v = -1$. To make the algorithm more readable, we introduce variable $\mathsf{Cid}_v = (\widehat{\mathsf{B}}_v, \mathsf{ph}_v, \mathsf{Bp}_v)$, called a *compact identifier* in the sequel. When meaningful, we use $\mathsf{Cid}_v^i = (\widehat{\mathsf{B}}_v, \mathsf{Bp}_v)$, where $i = \mathsf{ph}_v$.

Node v can trivially detect an error (see predicate $\mathtt{ErT}(v)$) whenever its compact identifier does not match its global identifier, or its phase is greater than $\widehat{\mathsf{B}}_v$.

$$\mathtt{ErT}(v) \equiv \big[\mathsf{Cid}_v \neq (\mathtt{Bit}_v(1), \mathsf{ph}_v, \mathtt{Bit}_v(\mathsf{ph}_v))\big] \vee (\mathsf{ph}_v > \widehat{\mathsf{B}}_v) \tag{1}$$

Moreover, the phases of neighboring nodes must be close enough: a node's phase may not be more than 1 ahead or behind any of its neighbors; also a node may not have a neighbor ahead and another behind. Predicate $\mathtt{SErB}(v, S)$ captures these conditions, where $S(v)$ denotes a subset of neighbors of v. The set S should be understood as an input provided by an upper layer algorithm.

$$\mathtt{SErB}(v, S) \equiv \Big(\exists u, w \in S(v) : \big(\mathsf{ph}_u > \mathsf{ph}_v + 1\big) \vee \big(\mathsf{ph}_u < \mathsf{ph}_v - 1\big) \vee \big(|\mathsf{ph}_u - \mathsf{ph}_w| = 2\big)\Big) \tag{2}$$

If v detects an error through $\mathtt{ErT}(v)$ or $\mathtt{SErB}(v, S)$, it resets its compact identifier to its first phase value (see command $\mathcal{R}esetCid(v)$). In a talkative process, node identifiers are published (though compact identifiers) infinitely often. So, when node v and all its active neighbors have reached the maximum phase (*i.e.* $\mathsf{ph}_v = \widehat{\mathsf{B}}_v$), v goes back to phase one. Then, if v has $\mathsf{ph}_v = \widehat{\mathsf{B}}_v$ and an active neighbor u has $\mathsf{ph}_u = 1$, it is not an error. But if v has $\mathsf{ph}_v = 1$, one active

neighbor u has $\mathsf{ph}_u = \widehat{\mathsf{B}}_v$, and another active neighbor w has $\mathsf{ph}_w > 1$, then an error is detected.

$$\begin{aligned}\texttt{TErB}(v,S) \equiv \Big(\exists u,w \in S(v) : \big[(1 < \mathsf{ph}_v < \widehat{\mathsf{B}}_v) \wedge ((\mathsf{ph}_u > \mathsf{ph}_v + 1) \vee (\mathsf{ph}_u < \mathsf{ph}_v - 1))\big] \vee \\ \big[(\mathsf{ph}_v = \widehat{\mathsf{B}}_v) \wedge ((\mathsf{ph}_u > 1) \vee (\mathsf{ph}_u < \widehat{\mathsf{B}}_v - 1) \vee ((\mathsf{ph}_u = \mathsf{ph}_v - 1) \wedge (\mathsf{ph}_w = 1)))\big] \vee \\ \big[(\mathsf{ph}_v = 1) \wedge ((\mathsf{ph}_u > 2) \vee (\mathsf{ph}_u < \widehat{\mathsf{B}}_v) \vee ((\mathsf{ph}_u = \widehat{\mathsf{B}}_v) \wedge (\mathsf{ph}_w = 2)))\big]\Big)\end{aligned} \tag{3}$$

If v detects an error through $\texttt{ErT}(v)$, $\texttt{SErB}(v,S)$ or $\texttt{TErB}(v,S)$, it resets its compact identifier to its first phase value:

$$\texttt{Er}(v,S) \equiv \texttt{ErT}(v) \vee SErB(v,S) \vee \texttt{TErB}(v,S) \tag{4}$$

$$\mathcal{R}esetCid(v) : \mathsf{Cid}_v^1 := (\widehat{\mathsf{B}}_v, \mathsf{ph}_v, \mathsf{Bp}_v) = (\texttt{Bit}_v(1), 1, \texttt{Bit}_v(1)) \tag{5}$$

This may trigger similar actions at neighbors in S, so that all such errors eventually disappear.

The compact identifier of u is smaller (respectively greater) than the compact identifier of v, if the most significant bit-position of u is smaller (respectively greater) than the most significant bit-position of v, or if the most significant bit-position of u is equal to the most significant bit-position of v, u and v are in the same phase, and the bit-position of u is smaller (respectively greater) than the bit-position of v:

$$\mathsf{Cid}_u^i <_c \mathsf{Cid}_v^i \equiv (\widehat{\mathsf{B}}_u < \widehat{\mathsf{B}}_v) \vee \big((\widehat{\mathsf{B}}_v = \widehat{\mathsf{B}}_u) \wedge (\mathsf{Bp}_u < \mathsf{Bp}_v)\big) \tag{6}$$

When two nodes u and v have the same most significant bit-position and the same bit position at phase $i < \widehat{\mathsf{B}}_v$, they are possibly equal with respect to compact identifiers (denoted by $\simeq_c$).

$$\mathsf{Cid}_u^i \simeq_c \mathsf{Cid}_v^i \equiv (i < \widehat{\mathsf{B}}_v) \wedge \big((\widehat{\mathsf{B}}_v = \widehat{\mathsf{B}}_u) \wedge (\mathsf{Bp}_u = \mathsf{Bp}_v)\big) \tag{7}$$

Finally, two nodes u and v have the same compact identifier (denoted by $=_c$) if their phase reaches the size of the binary representation of the identifier of the two nodes, and their last bit-position is the same.

$$\mathsf{Cid}_u^i =_c \mathsf{Cid}_v^i \equiv (i = \widehat{\mathsf{B}}_v = \widehat{\mathsf{B}}_u) \wedge (\mathsf{Bp}_u = \mathsf{Bp}_v) \tag{8}$$

The predicates $\texttt{SPh}^+(v)$ and $\texttt{TPh}^+(v,S)$ check if a node v can increases its phase (or restarts Cid_v), the first one is dedicated to the silent protocols, the second one is dedicated to the talkative protocols. The command $\mathcal{I}ncPh(v)$ is dedicated for increasing phases or restarting Cid_v. Last, the command $\mathcal{O}pt$ assigns at a node v the minimum (or maximum) compact identifier in the subset of neighbors $S(v)$. We have now, all the principals ingredients to use compact identifiers.

Predicate $\texttt{SPh}^+(v)$ is true if for every node u in $S(v)$, either $\mathsf{Cid}_u^i \simeq_c \mathsf{Cid}_v^i$, or $\mathsf{ph}_u = \mathsf{ph}_v + 1$.

$$\texttt{SPh}^+(v,S) \equiv \forall u \in S(v) : (\mathsf{Cid}_u^i \simeq_c \mathsf{Cid}_v^i) \vee (\mathsf{ph}_u = \mathsf{ph}_v + 1) \tag{9}$$

Similarly, $\texttt{TPh}^+(v,S)$ is true if for every node u in $S(v)$, either $\mathsf{Cid}_u^i =_c \mathsf{Cid}_v^i$, or $\mathsf{ph}_u = 1$. Remark that when the self-stabilizing algorithm is talkative, when the phase reaches the maximum, the publication restart at the first phase. As a consequence, the next phase to compare when a node reaches the maximum phase is the phase 1.

$$\texttt{TPh}^+(v,S) \equiv \texttt{SPh}^+(v,S) \vee \forall u \in S(v) : (\mathsf{ph}_v = \widehat{\mathsf{B}}_v) \wedge \big((\mathsf{Cid}_u^i =_c \mathsf{Cid}_v^i) \vee (\mathsf{ph}_u = 1)\big) \tag{10}$$

When $\texttt{TPh}^+(v,S)$ or $\texttt{SPh}^+(v)$ is true, v may increase its phase:

$$\mathcal{I}ncPh(v) : \mathsf{Cid}_v := \begin{cases} (\widehat{\mathsf{B}}_v, \mathsf{ph}_v + 1, \texttt{Bit}_v(\mathsf{ph}_v + 1)) & \text{if } \mathsf{ph}_v < \widehat{\mathsf{B}}_v \\ (\widehat{\mathsf{B}}_v, 1, \texttt{Bit}_v(1)) & \text{if } \mathsf{ph}_v = \widehat{\mathsf{B}}_v \end{cases} \tag{11}$$

In some case, we need to compute the minimum or the maximum on compact identifiers. Let f denote a function that is either minimum or maximum. Let us denote by $\texttt{C}\widehat{\mathsf{B}}(v,S,f)$ the minimum or the maximum most significant bit of nodes in $S(v)$.

$$\texttt{C}\widehat{\mathsf{B}}(v,S,f) = f\{\widehat{\mathsf{B}}_w : w \in S(v)\} \tag{12}$$

To compare compact identifiers, one must always refer to the same phase; we always consider the minimum phase for nodes in $S(v)$.

$$\texttt{CPh}(v,S,f) = \min\{\mathsf{ph}_w : w \in S(v), \widehat{\mathsf{B}}_w = \texttt{C}\widehat{\mathsf{B}}(w,S,f)\} \tag{13}$$

Finally, we compute the minimum or the maximum bit position.

$$\texttt{CBp}(v,S,f) = f\{\mathsf{Bp}_w : w \in S(v), \mathsf{ph}_w = \texttt{CPh}(w,S,f)\} \tag{14}$$

Predicate $\texttt{MinCid}(v,S)$ checks if Cid_v is equal to the minimum among nodes in $S(v)$:

$$\texttt{MinCid}(v,S) \equiv \big(\mathsf{Cid}_v = (\texttt{C}\widehat{\mathsf{B}}(v,S,\min), \texttt{CPh}(v,S,\min), \texttt{CBp}(v,S,\min))\big) \tag{15}$$

The predicate $\texttt{MaxCid}(v,S)$ does the same for the maximum:

$$\texttt{MaxCid}(v,S) \equiv \big(\mathsf{Cid}_v = (\texttt{C}\widehat{\mathsf{B}}(v,S,\max), \texttt{CPh}(v,S,\max), \texttt{CBp}(v,S,\max))\big) \tag{16}$$

Node v may use $\mathcal{O}pt$ to assign its local variables the minimum (or maximum) compact identifier in $S(v)$.

$$\mathcal{O}pt(v,S,f) : \begin{cases} \widehat{\mathsf{B}}_v := \texttt{C}\widehat{\mathsf{B}}(v,S,f) \\ \mathsf{ph}_v := \texttt{CPh}(v,S,f) \\ \mathsf{Bp}_v := \texttt{CBp}(v,S,f) \end{cases} \tag{17}$$

We have now, all the ingredients to use compact identifiers.

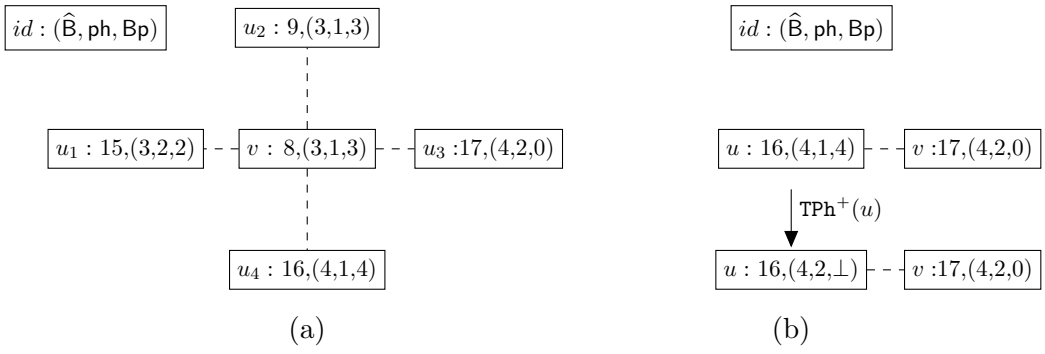

Figure 2: On the left subfigure (a), the following holds: Cid_v and Cid_{u_1} are incomparable; $Cid_v^1 \simeq_c Cid_{u_2}^1$; $Cid_v <_c Cid_{u_3}$;$Cid_v <_c Cid_{u_4}$, and $\texttt{MaxCid}(v, N(v)) = (4,1,4)$. On the right subfigure (b), nodes u and v are incomparable, so u increases its phase.

3.2 Silent self-stabilizing distance-2 coloring

In this section, we provide a new solution to assign colors that are unique up to distance two (and bounded by a polynom of the graph degree) in any graph. Those colors are meant to efficiently implement the pointer-to-neighbor mechanism that otherwise requires $\Omega(\log n)$ bits of memory per node. The remaining of the section is organized as follows: Subsection 3.2.1 presents high level concepts of our solution, Subsection 3.2.2 lists functions and predicates used in the algorithm, Subsection 3.2.3 formally presents the algorithm, while Subsection 3.2.4 is dedicated to establishing its proof of correctness.

3.2.1 Self-stabilizing algorithm high level description

Our solution uses compact identifiers to reduce memory usage. When a node v has the same color as (at least one of) its neighbors, then if the node v has the smallest conflicting color in its neighborhood and is not the biggest identifier among conflicting nodes, then v changes its color. To make sure a fresh color is chosen by v, all nodes publish the maximum color used by their neighborhood (including themself). So, when v changes its color, it takes the maximum advertised color plus one. Conflicts at distance two are resolved as follow: let us consider two nodes u and v in conflict at distance two, and let w be (one of) their common neighbor; as w publishes the color of u and v, it also plays the role of a relay, that is, w computes and advertises the maximum identifiers between u and v, using the compact identifiers mechanisms that were presented above; a bit by bit, then, if v has the smallest identifier, it changes its color to a fresh one. To avoid using too many colors when selecting a fresh one, all changes of colors are made modulo an upper bound on the number of neighbors at distance 2, which is computed locally by each node.

In our self-stabilizing coloring algorithm, called **C-Color**, each node v maintains a color variable denoted by c_v and a degree variable denoted by δ_v. A variable $\check{\mathsf{c}}_v$ stores the minimum color in conflict in its neighborhood (including itself). The variable $\hat{\mathsf{c}}_v$ stores the maximum color observed in its neighborhood. We call v a *player* node when v has the minimum color in conflict. Also, we call u a *relay* node when u does not have the minimum color in conflict, yet at least two of its neighbors have the minimum color in conflict. The nodes continuously update their variables according to the minimum and maximum colors published by their neighbors. A player node whose compact identifier is smaller than the compact identifier of its neighbors (be them players or relay nodes) becomes a *loser* node and changes its color. If all the neighborhood of a player node p has the same phase and the same compact identifier, then p increases its phase, until it becomes a loser or no conflict remains in p's neighborhood. When a player node and its neighborhood reach the maximum phase, they restart at the first phase. A relay node continuously takes the value of the greatest compact identifier of its player neighbors, for the purpose that at least one of them becomes a loser. Note that, a node may alternate between being a relay and a player, until the coloring is complete. This algorithm is silent, once the system reaches a distance two coloring, the color variables remain the same.

3.2.2 Functions, predicates and actions used by algorithm C-Color

Note that all rules are exclusive, because a node v cannot be both $\texttt{Player}(v)$ and $\texttt{Relay}(v)$. Let us now describe the functions, predicates and actions use by algorithm **C-Color**. Remember that $N[v] = N(v) \cup \{v\}$. Function $\Delta(v)$ returns the maximum degree between v and its neighbors, and is used to define the range $[1, \Delta(v)^2 + 1]$ of authorized colors for a node v:

$$\Delta(v) = \max\{\delta_u : u \in N[v]\} \tag{18}$$

The function $\texttt{mC}(v)$ returns the minimum color in conflict at distance one and two :

$$\texttt{mC}(v) := \min\{\mathsf{c}_u : u, w \in N[v] \wedge (u \neq w) \wedge (\mathsf{c}_u = \mathsf{c}_w)\} \tag{19}$$

The function $\texttt{MC}(v)$ returns the maximum color used at distance one :

$$\texttt{MC}(v) := \max\left\{\mathsf{c}_u : u \in N[v]\right\} \tag{20}$$

The predicate $\texttt{Bad}(v)$ is true if v has not yet set the right value for either $\texttt{mC}(v)$ or $\texttt{MC}(v)$. Moreover, this predicate checks if v's compact identifier matches its global identifier (see predicate 1: $\texttt{ErT}(v)$) and if the phases of the subset of v's neighbors $\texttt{Oth}(v)$ are coherent with v (see predicate 2: $\texttt{SErB}(v, \texttt{Oth}(v))$.

$$\texttt{Bad}(v) \equiv (\check{\mathsf{c}}_v \neq \texttt{mC}(v)) \vee (\hat{\mathsf{c}}_v \neq \texttt{MC}(v)) \vee (\texttt{ErT}(v) \vee \texttt{SErB}(v, \texttt{Oth}(v)) \tag{21}$$

The predicate $\texttt{Player}(v)$ is true if v has the minimum color in conflict (announced by its neighbors or by itself). Observe that the conflict may be at distance one or two:

$$\texttt{Player}(v) \equiv (\mathsf{c}_v = \min\{\check{\mathsf{c}}_u, u \in N[v]\}) \tag{22}$$

The predicate $\texttt{Relay}(v)$ is true if v does not have the minimum color in conflict, and at least two of its neighbors have the minimum color in conflict:

$$\texttt{Relay}(v) \equiv (\check{\mathsf{c}}_v \neq \mathsf{c}_v) \wedge (\check{\mathsf{c}}_v = \min\{\check{\mathsf{c}}_u, u \in N[v]\}) \wedge (\exists u, w \in N(v), (\check{\mathsf{c}}_v = \mathsf{c}_u) \wedge (\check{\mathsf{c}}_v = \mathsf{c}_w)) \tag{23}$$

The function $\texttt{PlayR}(v)$ returns the subset of v's neighbors that have the minimum color in conflict, when v is a relay node:

$$\texttt{PlayR}(v) := \{u : u \in N(v) \wedge \mathsf{c}_u = \check{\mathsf{c}}_v\} \tag{24}$$

The function $\texttt{Oth}(v)$ returns the subset of v's neighbors that are in conflict with v at distance one, or the set of relay nodes for the conflict at distance two, when v has $\texttt{Player}(v)$ equal to *true*:

$$\texttt{Oth}(v) := \{u : u \in N(v) \wedge \mathsf{c}_u = \mathsf{c}_v\} \cup \{u : u \in N(v) \wedge \check{\mathsf{c}}_u = \mathsf{c}_v\} \tag{25}$$

The predicate $\texttt{Loser}(v)$ is true whenever a competing player of v has a greater bit position at the same phase. A node whose identifier is maximum among competitors does not change its color, but any losing competitor does.

$$\texttt{Loser}(v) \equiv \exists u \in \texttt{Oth}(v) : \mathsf{Cid}_v^i <_c \mathsf{Cid}_u^i \tag{26}$$

The predicate $\texttt{RUp}(v)$ is true if a relay node is not according to its player neighbors, like we decide to change the color of the node with the minimum identifier the relay node stores the maximum compact identifier of its player neighbors:

$$\texttt{RUp}(v, \texttt{PlayR}(v)) \equiv \mathsf{Cid}_v \neq_c \texttt{MaxCid}(v, \texttt{PlayR}(v)) \tag{27}$$

The action $\mathcal{U}pdate(v)$ updates the variables $\check{\mathsf{c}}_v, \hat{\mathsf{c}}_v$ and resets the variables relatives to the identifier (see command $\mathcal{R}esetCid$(v) in equation 5).

$$\mathcal{U}pdate(v) : \check{\mathsf{c}}_v := \texttt{mC}(v); \hat{\mathsf{c}}_v := \texttt{MC}(v); \mathcal{R}esetCid(v); \tag{28}$$

When a node change its color, it takes the maximum color at distance one and two plus one modulo $\Delta(v)^2 + 1$, and then add one to assign colors in the range $[1, \ldots, \Delta(v)^2 + 1]$.

$$\mathcal{N}ewcolor(v) : \mathsf{c}_v := \left((\max\{\texttt{MC}_u : u \in N[v]\} + 1) \mod \Delta(v)^2 + 1\right) + 1; \tag{29}$$

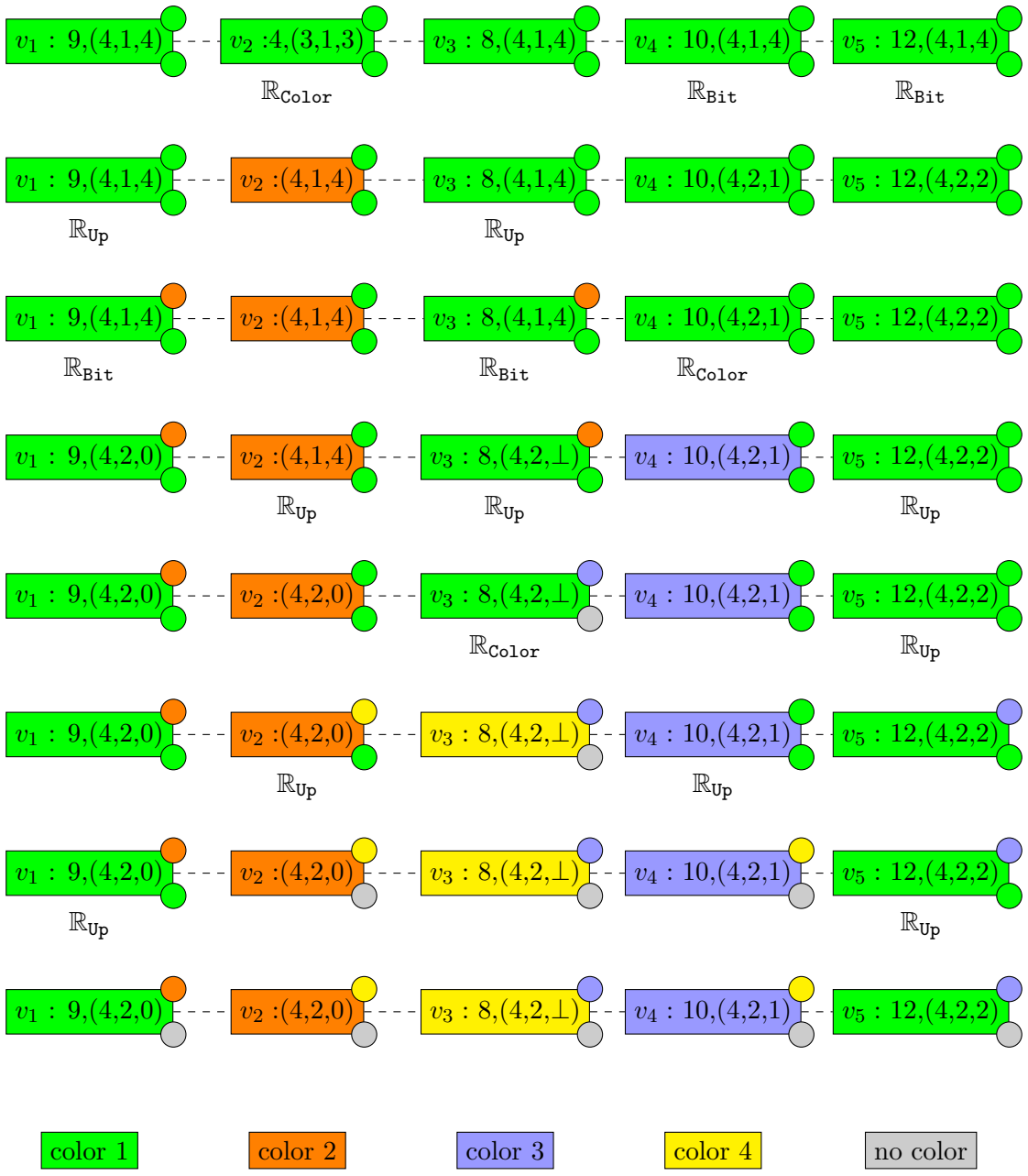

Figure 3: An example execution of our coloring scheme is presented for a line network of size 5. The color of the main rectangle denotes the node's color (variable c_v), the upper left disc color denotes the maximum color used at distance 1 (variable $\widehat{\mathsf{c}}_v$), the lower left disc color denotes the minimum color in conflict at distance 1 or 2 (variable $\check{\mathsf{c}}_v$). The light gray color denotes the absence of any color. The rule immediately under a node denotes which rule is executed to reach the next configuration of the execution.

3.2.3 Algorithm C-Color

The rule $\mathbb{R}_\Delta$ assures that the degree variable is equal to the degree of the node. Each node v must maintain its color in range $[1, \Delta(v)^2 + 1]$ to satisfy the memory requirements of our protocol, where $\Delta(v)$ is a function that returns the maximum degree of its neighborhood (including itself).

Whenever v's color exceeds its expected range, rule $\mathbb{R}^+_\Delta$ resets the color to one. Rule $\mathbb{R}_{\texttt{Up}}$ is dedicated to updating the variables of v whenever they do not match the observed neighborhood of v (see $\texttt{Bad}(v)$), or when a player node has an erroneous phase variable when comparing its identifier with another player node (see function $\texttt{Oth}(v)$). In both cases, the v computes the minimum and maximum color and resets its compact identifier variable (see command $\mathcal{U}pdate(v)$ 28). The rule $\mathbb{R}_{\texttt{Color}}$ increases the color of the node v but maintains the color in some range (see command $\mathcal{N}ewcolor(v)$ 29), when v has the minimum color in conflict and the minimum identifier. The rule $\mathbb{R}_{\texttt{Bit}}$ increases the phase of v, when v is a player and does not have the minimum identifier at the selected phase. The rule $\mathbb{R}_{\texttt{Relay}}$ updates the identifier variable when v is a relay node.

Algorithm 1: C-Color

$$
\begin{array}{rcll}
\mathbb{R}_\Delta & : & (\delta_v \neq \deg(v)) & \longrightarrow \quad \delta_v := \deg(v); \\
\mathbb{R}^+_\Delta & : & (\delta_v = \deg(v)) \wedge (\texttt{c}_v > \Delta(v)^2) & \longrightarrow \quad \texttt{c}_v := 1; \\
\mathbb{R}_{\texttt{Up}} & : & (\delta_v = \deg(v)) \wedge (\texttt{c}_v \leqslant \Delta(v)^2) \wedge \texttt{Bad}(v) & \longrightarrow \quad \mathcal{U}pdate(v); \\
\mathbb{R}_{\texttt{Color}} & : & (\delta_v = \deg(v)) \wedge (\texttt{c}_v \leqslant \Delta(v)^2) \wedge \neg\texttt{Bad}(v) \wedge \texttt{Player}(v) \wedge \texttt{Loser}(v) \longrightarrow \mathcal{N}ewcolor(v); & \\
\mathbb{R}_{\texttt{Bit}} & : & (\delta_v = \deg(v)) \wedge (\texttt{c}_v \leqslant \Delta(v)^2) \wedge \neg\texttt{Bad}(v) \wedge \texttt{Player}(v) \wedge \neg\texttt{Loser}(v) \wedge \texttt{SPh}^+(v, \texttt{Oth}(v)) & \\
 & & \longrightarrow \quad \mathcal{I}ncPh(v); & \\
\mathbb{R}_{\texttt{Relay}} & : & (\delta_v = \deg(v)) \wedge (\texttt{c}_v \leqslant \Delta(v)^2) \wedge \neg\texttt{Bad}(v) \wedge \texttt{Relay}(v) \wedge \texttt{RUp}(v, \texttt{PlayR}(v)) & \\
 & & \longrightarrow \quad \mathcal{O}pt(v, \texttt{PlayR}(v), \max); &
\end{array}
$$

3.2.4 Correctness

Theorem 2. *Algorithm* **C-Color** *solves the vertex coloration problem at distance two using* $\Delta^2 + 1$ *colors in a silent self-stabilizing manner in graph, assuming the state model, and a distributed unfair scheduler. Moreover, if the* n *node identifiers are in* $[1, n^c]$*, for some* $c \geqslant 1$*, then* **C-Color** *uses* $O(\log \Delta + \log\log n)$ *bits of memory per node.*

In the details of lemmas that are presented in the sequel, we use predicates on configurations. These predicates are mean to be intermediate attractors towards a legitimate configuration (*i.e.*, a configuration with a unique leader). To establish that those predicates are indeed attractors, we use potential functions [5, 39, 40], that is, functions that map configurations to non-negative integers, and that strictly decrease after any algorithm step is executed. In the remaining of the paper, the potential functions we define closely match the proof arguments of the following Lemma/Theorem. That is, various invariants are defined for each property we wish to prove, and the potential function makes sure the output value decreases until the invariant is reached.

To avoid additional notations, we use sets of configurations to define predicates; the predicate should then be understood as the characteristic function of the set (that returns *true* if the configuration is in the set, and *false* otherwise).

Lemma 1. *Using a range of* $[1, \Delta(v)^2 + 1]$ *for colors at node* v *is sufficient to enable distance-2 coloring of the graph.*

Proof. In the worst case for the number of colors, all neighbors at distance one and two of v have different colors. Now, v has at most $\Delta(v)$ neighbors at distance one, each having $\Delta(v) - 1$ other neighbors than v. In total, v has at most $\Delta(v)^2 - \Delta(v)$ neighbors at distance up to two, each having a distinct color. Using a range of $[1, \Delta(v)^2 + 1]$ for v's color leaves at least $\Delta(v) + 1$ available colors for node v. □

Let $\mathsf{c}_v(\gamma)$ be the color of v in configuration γ.

Let $\lambda : \Gamma \times V \to \mathbb{N}$ be the following function:

$$\lambda(\gamma, v) = \begin{cases} 2 & \text{if } \delta_v \neq \deg(v) \in \gamma \\ 1 & \text{if } (\delta_v = \deg(v) \in \gamma) \wedge \mathsf{c}_v(\gamma) > \Delta(v)^2 + 1 \\ 0 & \text{otherwise} \end{cases}$$

Let $\Lambda : \Gamma \to \mathbb{N}$ be the following potential function:

$$\Lambda(\gamma) = \sum_{v \in V} \lambda(\gamma, v)$$

Now, let $\mathsf{c}_v(\gamma_0)$ be the color of v in configuration γ_0, γ_0 being defined as the configuration where $\Lambda(\gamma_0)$ reaches zero.

Let $\tau : \Gamma \times V \to \mathbb{N}$ be the following function:

$$\tau(\gamma, v) = \begin{cases} \Delta(v)^2 + 1 + \mathsf{c}_v(\gamma_0) - \mathsf{c}_v(\gamma) & \text{if } \mathsf{c}_v(\gamma) > \mathsf{c}_v(\gamma_0) \\ \mathsf{c}_v(\gamma_0) - \mathsf{c}_v(\gamma) & \text{otherwise} \end{cases}$$

Let $\mathcal{C} : \Gamma \to \mathbb{N}$ be the following potential function:

$$\mathcal{C}(\gamma) = \sum_{v \in V} \tau(\gamma, v)$$

We denote by γ' the configuration after activation of (a subset of) the nodes in $A_C(\gamma)$ where $A_C(\gamma)$ denotes the enabled nodes in γ due to rule $\mathbb{R}_{\texttt{Color}}$.We can now prove the following result: $\mathcal{C}(\gamma') < \mathcal{C}(\gamma)$ for every configuration γ where $A_C(\gamma)$ is not empty.

Lemma 2. *$\mathcal{C}(\gamma') < \mathcal{C}(\gamma)$ for every configuration γ such that $A_C(\gamma)$ is not empty.*

Proof. We consider a node $v \in A_C(\gamma)$. After executing rule $\mathbb{R}_{\texttt{Color}}$, v takes a color:

$$\mathsf{c}_v(\gamma') = \big((\max\{\texttt{MC}_u : u \in N[v]\} + 1) \mod \Delta(v)^2 + 1\big) + 1$$

As a consequence $\tau(\gamma, v)$ decreases by at least one, so $\mathcal{C}(\gamma') < \mathcal{C}(\gamma)$. □

Let $\psi : \Gamma \times V \to \mathbb{N}$ be the function defined by:

$$\psi(\gamma, v) = \begin{cases} n^3 & \text{if } \texttt{Bad}(v) \text{ is true} \\ (2\log n - \mathsf{ph}_v) \times (n^2 - \breve{\mathsf{c}}_v) & \text{if } \neg\texttt{Bad}(v) \wedge \texttt{Player}(v) \text{ is true} \\ 2(n^2 - \breve{\mathsf{c}}_v) & \text{if } \neg\texttt{Bad}(v) \wedge \texttt{Relay}(v) \wedge \neg\texttt{RUp}(v, \texttt{PlayR}(v)) \text{ is true} \\ (n^2 - \breve{\mathsf{c}}_v) & \text{if } \neg\texttt{Bad}(v) \wedge \texttt{Relay}(v) \wedge \texttt{RUp}(v, \texttt{PlayR}(v)) \text{ is true} \\ 0 & \text{otherwise} \end{cases}$$

Let $\Psi : \Gamma \to \mathbb{N}$ be the potential function defined by:

$$\Psi(\gamma) = \sum_{v \in V} \psi(\gamma, v)$$

Let $\Phi : \Gamma \to \mathbb{N}^3$ be the potential function defined by:

$$\Phi(\gamma) = (\Lambda(\gamma), \mathcal{C}(\gamma), \Psi(\gamma)).$$

The comparison between two configurations $\Phi(\gamma)$ and $\Phi(\gamma')$ is by using lexical order. We denote by $A(\gamma)$ the (subset of) enabled nodes (for any rule of our algorithm) in configuration γ. Note that the algorithm is stabilized when every node is neither a player nor a relay, that is the nodes have no conflict at distance one and two, when $\Psi(\gamma) = 0$. We define

$$\Gamma_{\mathcal{C}} = \{\gamma \in \Gamma : \Phi(\gamma) = 0\}$$

Lemma 3. *true* $\rhd \Gamma_{\mathcal{C}}$ *and* $\Gamma_{\mathcal{C}}$ *is closed.*

Proof. The function $\Lambda(\gamma)$ decreases by any execution of rules $\mathbb{R}_{\Delta}$ and $\mathbb{R}^{+}_{\Delta}$. Remark that $\deg(v)$ is considered a non corruptible local information, so once v has executed $\mathbb{R}_{\Delta}$, this rule remains disabled afterwards. Moreover, $\mathbb{R}_{\texttt{Color}}$ maintains the value of the color inferior (or equal) to $\Delta(v)^2+1$, and other rules modifying the color maintain this invariant. Hence, if the scheduler activates rules $\mathbb{R}_{\Delta}$ or $\mathbb{R}^{+}_{\Delta}$, we obtain $\Lambda(\gamma') < \Lambda(\gamma)$, otherwise if other rules are activated, then $\Lambda(\gamma') = \Lambda(\gamma)$. We already saw that, when the scheduler activates a node v for rule $\mathbb{R}_{\texttt{Color}}$, we obtain $\mathcal{C}(\gamma') < \mathcal{C}(\gamma)$. Overall, if the scheduler activates rules $\mathbb{R}_{\Delta}$, $\mathbb{R}^{+}_{\Delta}$, or $\mathbb{R}_{\texttt{Color}}$ we obtain $\Phi(\gamma') < \Phi(\gamma)$. We now consider the cases where the scheduler activates other rules.

First, we focus on rule $\mathbb{R}_{\texttt{Up}}$. Let us consider $A'(\gamma)$, the set of nodes enabled in configuration γ for this rule, and a node v such that $v \in A'(\gamma)$. In other words, v has $\texttt{Bad}(v) = \mathit{true}$ (see predicate 21). If v has $(\check{\texttt{c}}_v \neq \texttt{mC}(v)) \vee (\hat{\texttt{c}}_v \neq \texttt{MC}(v))$ in γ, then after activation of v, we obtain $\check{\texttt{c}}_v = \texttt{mC}(v)$ and $\hat{\texttt{c}}_v = \texttt{MC}(v)$ in γ' by the command $\mathcal{N}ewcolor(v)$ (see command 29), because $\texttt{mC}(v)$ and $\texttt{MC}(v)$ depend only on the color of the neighbors of v (see Function 19 and 20). The same argument applies for $(\texttt{ErT}(v) \vee \texttt{SErB}(v, \texttt{Oth}(v)))$, because $\texttt{Cid}_v$ is computed only with the identifier of v. So, after execution of $\mathbb{R}_{\texttt{Up}}$ by v, we obtain $\psi(\gamma', v) < \psi(\gamma, v)$, thanks to the execution of the command $\mathcal{U}pdate$(v) (see command 28) that assigns $false$ to $\texttt{Bad}(v)$. Remark that, if the color of the neighbors of v does not change, rule $\mathbb{R}_{\texttt{Up}}$ remains disabled. Now, if the color changes, $\Phi(\gamma)$ still decreases thanks to Lemma 2.

Let us consider now a configuration where the rule $\mathbb{R}_{\texttt{Up}}$ is disabled for every node. Rule $\mathbb{R}_{\texttt{Bit}}$ increases the phase of a player node, so after activation of this rule we obtain $\psi(\gamma', v) < \psi(\gamma, v)$. Executing rule $\mathbb{R}_{\texttt{Relay}}$ decreases also $\psi(\gamma', v)$ due to $\texttt{RUp}(v, \texttt{PlayR}(v))$ (see predicate 27) , because when all nodes in $\texttt{PlayR}(v)$ (see function 24) have increase their phases, ψ decreases for all v's neighbors. Note that, when a conflict for a color $\texttt{c}$ is resolved, a node can become a player node or a relay node for another color $\texttt{c}'$, but then $\texttt{c}' > \texttt{c}$. So $(n - \texttt{c}') < (n - \texttt{c})$, hence ψ decreases.

If a node has no conflict at distance one or two, it never changes its color, so once the system reaches a distance-2 coloring, it remains with the same coloring. □

Lemma 4. *Algorithm* **C-Color** *requires* $O(\max\{\log\Delta, \log\log n\})$ *bits of memory per node.*

Proof. The variables $\delta_v, \texttt{c}_v, \check{\texttt{c}}_v, \hat{\texttt{c}}_v$ take $O(\log\Delta)$ bits. The compact identifier $\texttt{Cid}_v$ takes $O(\log\log n)$ bits per node. □

Proof of Theorem 2. Direct by Lemma 1, Lemma 3 and Lemma 4. □

Lemma 5. *Algorithm* **C-Color** *converges in* $O(\Delta^{\Delta^2} n^3)$ *steps.*

Proof. Direct by the potential function $\Phi(\gamma)$. □

3.3 Cleaning a cycle or an impostor-rooted spanning tree

The graph G is supposed to be colored up to distance 2, thanks to our previous algorithm. To construct a spanning tree of G, each node v maintains a variable $\texttt{p}_v$ storing the color of v's parent ($\varnothing$ otherwise). The function $\texttt{Ch}(v)$ to return the subset of v's neighbors considered as its children (that is, each such node u has its p_u variable equal v's color). Note that the variable parent is managed by the algorithm of spanning tree-construction.

An error is characterized by the presence of inconsistencies between the values of the variables of a node v and those of its neighbors. In the process of a tree-construction, an error occurring at node v may have an impact on its descendants. For this reasons, after a node v detects an error, our algorithm cleans v and all of its descendants. The cleaning process is achieved by Algorithm **Freeze**, already presented in previous works [13, 11]. Algorithm **Freeze** is run

in two cases: cycle detection (thanks to predicate $\texttt{ErCycle}(v)$, presented in Subsection 3.4), and impostor leader detection (thanks to predicate $\texttt{ErST}(v)$, presented in Subsection 4). An impostor leader is a node that (erroneously) believes that it is a root.

When a node v detects a cycle or an impostor root, v deletes its parent. Simultaneously, v becomes a *frozen* node. Then, every descendant of v becomes frozen. Finally, from the leaves of the spanning tree rooted at v, nodes delete their parent and reset all variables that are related to cycle detection or tree-construction. So, this cleaning process cannot create a livelock. Algorithm **Freeze** is a silent self-stabilizing algorithm using $O(1)$ bits of memory per node.

It is important to note that a frozen node, or the child of a frozen node, does not participate in cycle detection or spanning tree-construction.

We now recall **Freeze** in Algorithm 2. This algorithm uses only one binary variable froz. This approach presents several advantages. After v detecting a cycle, the cycle is broken (v deletes its parent), and a frozen node cannot reach its own subtree, due to the cleaning process taking place from the leaves to the root. So, two cleaning processes cannot create a livelock.

Algorithm 2: Algorithm **Freeze**

$$
\begin{array}{lcll}
\mathbb{R}_{\texttt{Error}} & : & \texttt{ErCycle}(v) \vee \texttt{ErST}(v) & \longrightarrow \mathsf{froz}_v := 1, \mathsf{p}_v := \varnothing; \\
\mathbb{R}_{\texttt{Froze}} & : & \neg\texttt{ErCycle}(v) \wedge \neg\texttt{ErST}(v) \wedge (\mathsf{froz}_{\mathsf{p}_v} = 1) \wedge (\mathsf{froz}_v = 0) & \longrightarrow \mathsf{froz}_v := 1; \\
\mathbb{R}_{\texttt{Prun}} & : & \neg\texttt{ErCycle}(v) \wedge \neg\texttt{ErST}(v) \wedge (\mathsf{froz}_{\mathsf{p}_v} = 1) \wedge (\mathsf{froz}_v = 1) \wedge (\texttt{Ch}(v) = \varnothing) & \longrightarrow \mathit{Reset}(v);
\end{array}
$$

Theorem 3. *Algorithm* **Freeze** *deletes a cycle or an impostor-rooted sub spanning tree in n-nodes graph in a silent self-stabilizing manner, assuming the state model, and a distributed unfair scheduler. Moreover, Algorithm* **Freeze** *uses $O(1)$ bits of memory per node and converges in $O(n)$ steps.*

Proof of Theorem 3 is due to Blin et al. [11].

3.4 Silent self-stabilizing algorithm for cycle detection

We present in this subsection a self-stabilizing algorithm to detect cycles (possibly due to initial incorrect configuration) without using the classical method of computing the distance to the root. We first present our solution with the assumption of global identifiers (hence using $O(\log n)$ bits for an n-node network) and then using our compact identifier scheme postponed in subsection 3.4.3.

3.4.1 Silent Self-stabilizing algorithm with identifiers

The main idea to detect cycles is to use the uniqueness of the identifiers. We flow the minimum identifier up to the tree to the root, then if a node whose identifier is minimum receives its identifier, it can detect a cycle. Similarly, if a node v has two children flowing the same minimum identifier, v can detect a cycle. The main issue to resolve is when the minimum identifier that is propagated to the root does not exist in the network (that is, it results from an erroneous initial state).

The variable m_v stores the minimum identifier collected from the leaves to the root up to node v. We denote by E_v the minimum identifier obtained by v during the previous iteration of the protocol (this can be $\varnothing$). A node v may selects among its children the node u with the smallest propagated identifier stored in m_u, we call this child *kid* returned by the function $\mathsf{k}(v)$.

Predicate $\texttt{ErCycle}(v)$ is the core of our algorithm. Indeed, a node v can detect the presence of a cycle if it has a parent and if *(i)* one of its children publishes its own identifier, or *(ii)* two of its children publish the same identifier. Let us explain those conditions in more detail. We consider a spanning structure S, a node $v \in S$ and let u and w be two of its children. Suppose that v and u belong to a cycle $\texttt{C}$, note that, since a node has a single parent, w cannot belong to any cycle (see Figure 4). Let $\breve{\mathsf{m}}$ be the minimum identifier stored by any variable m_z such that z belongs to S. So, z is either in $\texttt{C}$, or in the subtree rooted to w, denoted by $\texttt{T}_w$.

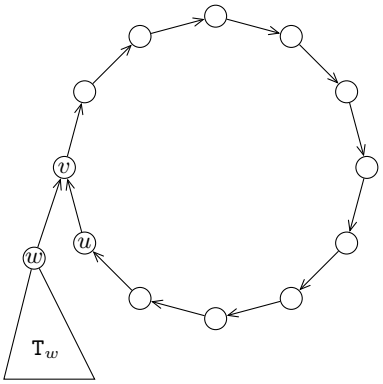

Figure 4: Spanning structure

First, let us consider the case where $\breve{\mathsf{m}}$ is stored in $\texttt{T}_w$. As any node selects the minimum for flowing the m upstream, there exists a configuration γ where $\mathsf{m}_w = \breve{\mathsf{m}}$, and a configuration $\gamma' > \gamma$ where $\mathsf{m}_u = \breve{\mathsf{m}}$. In γ', v can detect an error, due to the uniqueness of identifier, it is not possible for two children of v to share the same value when there is no cycle.

Now, let us suppose that $\breve{\mathsf{m}}$ is in $\texttt{C}$, and let v' be the node with the smallest identifier in $\texttt{C}$, so $\mathsf{m}_{v'} = \breve{\mathsf{m}}$ or $\mathsf{m}_{v'} \neq \breve{\mathsf{m}}$ ($\mathsf{m}_{v'} \neq \breve{\mathsf{m}}$ means that the identifier $\breve{\mathsf{m}}$ does not exist in $\texttt{C}$.) If $\mathsf{m}_{v'} = \breve{\mathsf{m}}$, as any node selects the minimum for flowing the m upstream, there exists a configuration γ where $\mathsf{m}_{u'} = \breve{\mathsf{m}}$ and u' is the child of v' involved in $\texttt{C}$, then v' can detect an error. Indeed, due to the uniqueness of identifier, it is not possible that one of its children store its identifier when there is no cycle. The remaining case is when $\mathsf{m}_{v'} \neq \breve{\mathsf{m}}$. In this case, as any node selects the minimum for flowing the m upstream, there exists a configuration γ where $\mathsf{m}_z = \breve{\mathsf{m}}$, with z belonging to $\texttt{C}$. When a node v, its parent and one of its children share the same minimum, they restart the computation of the minimum identifier. For this purpose, they put their own identifier in the m variable. To avoid livelock, they also keep track of the previous $\breve{\mathsf{m}}$ in variable E_v. Now $\breve{\mathsf{m}} = \mathsf{m}_{v'}$, so the system reaches the first case. Note that the variable E_v blocks the live-lock but also the perpetual restart of the nodes, as a result of this, a silent algorithm. Moreover, a node v collects the minimum identifier from the leaves to the root, if m_v contains an identifier bigger than the identifier of the node v, then v detects an error. The same holds, when v has a m_v smaller than m_u with u children of v, since the minimum is computed between $\mathsf{m}_{\mathsf{k}(v)}$ and its own identifier.

$$\texttt{ErCycle}(v) \equiv (\mathsf{p}_v \neq \varnothing) \wedge \Big((\mathsf{m}_{\mathsf{k}(v)} = \text{ID}_v) \vee (\exists (u,w) \in \texttt{Ch}(v) : \mathsf{m}_u = \mathsf{m}_w) \vee (\mathsf{m}_v > \text{ID}_v) \vee \big((\mathsf{m}_v \neq \text{ID}_v) \wedge (\mathsf{m}_v < \mathsf{m}_{\mathsf{k}(v)})\big)\Big)$$

Our algorithm only contains three rules. The rule $\mathbb{R}_{\texttt{Min}}(v)$ updates the variable m_v if the variable m_u of a child u is smaller, nevertheless this rule is enabled if and only if the variable E_v does not contain the minimum m_u published by the child. When v and its relatives have the same minimum, v declares its intent to restart a minimum identifier computation by erasing its current (and storing it in E_v). The rule $\mathbb{R}_{\texttt{Start}}(v)$ is dedicated to declaring its intent to restart. When all its neighbors have the same intent, v can restart (see rule $\mathbb{R}_{\texttt{ID}}(v)$).

An example execution of Algorithm **Break** is depicted in Figure 5.

3.4.2 Correctness of the algorithm Break

Theorem 4. *Algorithm* **Break** *solves the detection of cycle in n-node graph in a silent self-stabilizing manner, assuming the state model, and a distributed unfair scheduler. Moreover, if the n node identifiers are in $[1, n^c]$, for some $c \geqslant 1$, then algorithm* **Break** *uses $O(\log n)$ bits of memory per node.*

Algorithm 3: Algorithm **Break** For node v with $\neg\texttt{ErCycle}(v)$

$$\begin{array}{lll}
\mathbb{R}_{\texttt{Min}} & : & (\mathsf{m}_v > \mathsf{m}_{\mathsf{k}(v)}) \wedge (\mathsf{E}_v \neq \mathsf{m}_{\mathsf{k}(v)}) \longrightarrow \mathsf{m}_v := \mathsf{m}_{\mathsf{k}(v)}; \\
\mathbb{R}_{\texttt{Start}} & : & (\mathsf{m}_{\mathsf{p}_v} = \mathsf{m}_v = \mathsf{m}_{\mathsf{k}(v)}) \wedge (\mathsf{E}_v \neq \mathsf{m}_v) \longrightarrow \mathsf{E}_v := \mathsf{m}_v; \\
\mathbb{R}_{\texttt{ID}} & : & (\mathsf{E}_{\mathsf{p}_v} = \mathsf{E}_v = \mathsf{E}_{\mathsf{k}(v)} = \mathsf{m}_v) \wedge (\mathsf{m}_v \neq \mathrm{ID}_v) \longrightarrow \mathsf{m}_v := \mathrm{ID}_v;
\end{array}$$

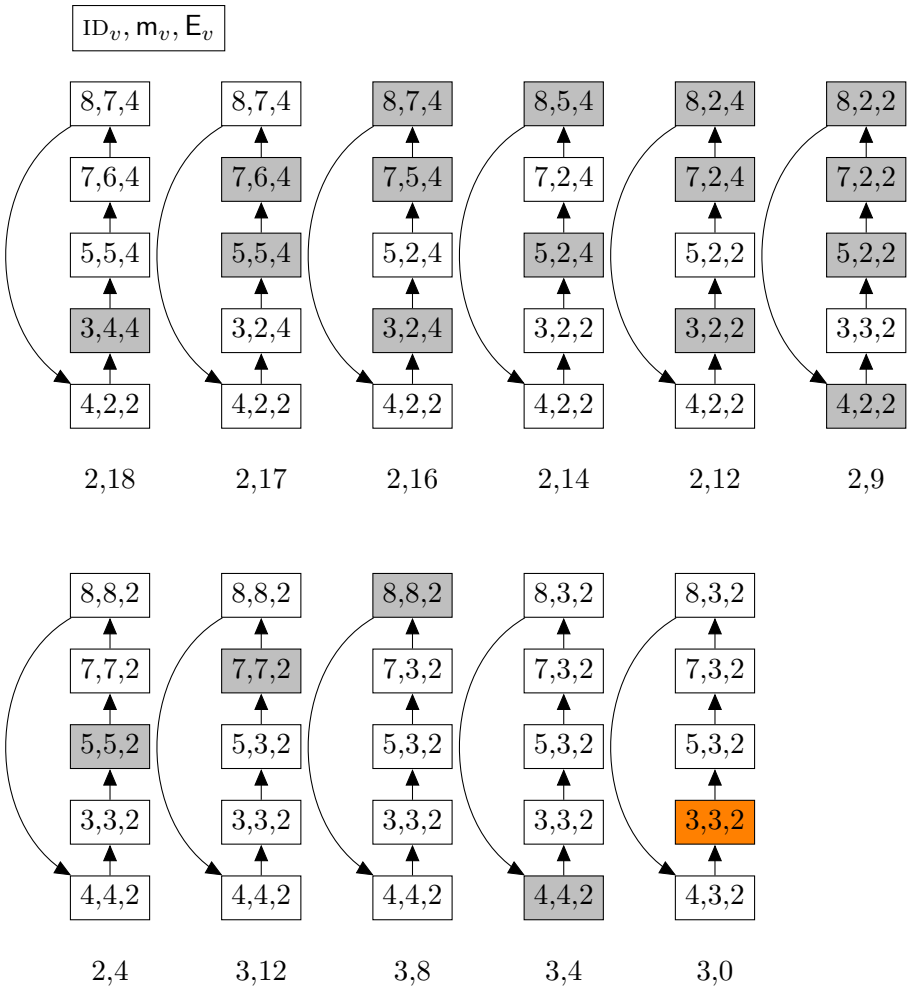

Figure 5: An example execution for Algorithm **Break** is presented on a ring shaped network. Only id, m_v, and E_v are represented for each node. Nodes colored in light gray are activated in each step by the scheduler. The node colored in orange detects a cycle. Only one of the many possible execution is represented. Below each configuration γ, we also indicate the value of the smallest identifier i such as $\beta(\gamma, i) > 0$, along with the value of the corresponding $\beta(\gamma, i)$ (See Section 3.4.2 for the definition of β.)

Let i be a natural integer in $[1, n^c]$, representing a possible node identifier. Then, let us denote by $D(v,i)$ the set of nodes on the path between v and w, where w is the nearest descendant of v ($w \neq v$) such that $\mathsf{m}_w = i$, if such a node exists. We suppose that every node u in $D(v,i)$ has $\texttt{ErCycle}(u) = \mathit{false}$. The value i can improve the value m_v if and only every node u in $D(v,i)$ has a $\mathsf{m}_u > i$ and $\mathsf{E}_u \neq i$. Also, if $\mathsf{E}_u = i$, the value vanishes during the execution, note that may be $u = v$. Predicate `Improve` captures this fact. Predicate `Improve` is defined recursivly as, to improve a node, its descendants must be improved first. Only when no descendent can be improved is the node itself bound for action.

$$\begin{aligned}\texttt{Improve}(\gamma, v, i) \equiv \big(\forall u \in D(v,i) : (\mathsf{m}_u > i)\big) \wedge \big[\big(\forall u \in D(v,i) : \mathsf{E}_u \neq i\big) \vee \\ \big(\exists u \in D(v,i) : (\mathsf{E}_u = i) \wedge \texttt{Improve}(\gamma, u, i)\big)\big]\end{aligned} \tag{30}$$

Let $\alpha : \Gamma \times V \times \mathbb{N} \to \mathbb{N}$ be the function defined by:

$$\alpha(\gamma, v, i) = \begin{cases} 4 & \text{if } (\mathsf{m}_v > i) \wedge (\mathsf{E}_v \neq i) \wedge \texttt{Improve}(\gamma, v, i) \\ 3 & \text{if } (\mathsf{m}_v = i) \wedge (\mathsf{E}_v \neq i) \wedge \texttt{Improve}(\gamma, v, i) \\ 2 & \text{if } (\mathsf{m}_v = i) \wedge (\mathsf{E}_v = i) \wedge \texttt{Improve}(\gamma, v, i) \\ 1 & \text{if } (\mathsf{m}_v > i) \wedge (\mathsf{E}_v = i) \wedge \texttt{Improve}(\gamma, v, i) \\ 0 & \text{if } \neg\texttt{Improve}(\gamma, v, i) \end{cases}$$

Let $\beta : \Gamma \times \mathbb{N} \to \mathbb{N}$ be the function defined by:

$$\beta(\gamma, i) = \sum_{v \in V} \alpha(\gamma, v, i)$$

Figure 5 depicts the evolution of the the values returned by β for the minimum identifier i such that $\beta(\gamma, i) > 0$.

Let $\Xi : \Gamma \to \mathbb{N}^{IdMax}$ be the function defined by:

$$\Xi(\gamma) = (\beta(\gamma, 1), \ldots, \beta(\gamma, IdMax))$$

The comparaison between two configurations $\Xi(\gamma)$ and $\Xi(\gamma')$ is performed using lexical order. In the following, $\mathsf{m}_v(\gamma)$ denotes the variable m_v in configuration γ. Note that the algorithm is stabilized when no value i can improve the value stored in m_v, that is when $\Xi(\gamma) = 0$. We define

$$\Gamma_{\mathcal{B}} = \{\gamma \in \Gamma : \Xi(\gamma) = 0\}$$

Lemma 6. *$true \rhd \Gamma_{\mathcal{B}}$ and $\Gamma_{\mathcal{B}}$ is closed.*

Proof. The starting predicate being *true*, all possible configurations are taken into consideration. We simply review the effect of executing the three rules of the algorithm:

- Rule $\mathbb{R}_{\texttt{Start}}(v)$: $\mathsf{m}_v(\gamma) = \mathsf{m}_v(\gamma')$, so for $i < \mathsf{m}_v(\gamma')$, we have $\beta(\gamma', i) = \beta(\gamma, i)$. Note that for $i > \mathsf{m}_v(\gamma)$, v has no effect on $\beta(\gamma, i)$ and $\beta(\gamma', i)$. Now, $\beta(\gamma, \mathsf{m}_v) = 3$ because $\mathbb{R}_{\texttt{Start}}(v)$ is enabled for v only if $(\mathsf{E}_v \neq \mathsf{m}_v)$, and $\beta(\gamma', \mathsf{m}_v) = 2$ because we have $(\mathsf{E}_v = \mathsf{m}_v)$, thus $\beta(\gamma, \mathsf{m}_v) = 3 > \beta(\gamma', \mathsf{m}_v) = 2$. So, if the scheduler activates v with rule $\mathbb{R}_{\texttt{Start}}(v)$, we obtain $\Xi(\gamma') < \Xi(\gamma)$.
- Rule $\mathbb{R}_{\texttt{ID}}$:
  - $i < \mathsf{m}_v(\gamma)$: $\beta(\gamma', i) = \beta(\gamma, i)$ because $\text{ID}_v > \mathsf{m}_v(\gamma)$ (otherwise an error is detected). Also, if i can improve $\mathsf{m}_v(\gamma)$, it can also improve $\mathsf{m}_v(\gamma')$.

– $\mathsf{m}_v(\gamma)$**:** Rule $\mathbb{R}_{\texttt{ID}}$ needs $\mathsf{E}_v(\gamma) = \mathsf{m}_v(\gamma)$, so $\beta(\gamma, \mathsf{m}_v(\gamma)) = 2$. Now, we have $\mathsf{E}_v(\gamma') = \mathsf{m}_v(\gamma) \neq \mathsf{m}_v(\gamma')$, and we obtain $\beta(\gamma, \mathsf{m}_v(\gamma)) = 2 > \beta(\gamma', \mathsf{m}_v(\gamma)) = 1$.

As a consequence, $\beta(\gamma', i) = \beta(\gamma, i)$ for $i < \mathsf{m}_v(\gamma)$ and $\beta(\gamma', \mathsf{m}_v(\gamma)) < \beta(\gamma, \mathsf{m}_v(\gamma))$. So, if the scheduler activates only v for rule $\mathbb{R}_{\texttt{ID}}(v)$, we obtain $\Xi(\gamma') < \Xi(\gamma)$.

- Rule $\mathbb{R}_{\texttt{Min}}(v)$: $\mathsf{m}_v(\gamma') < \mathsf{m}_v(\gamma)$ and $\mathsf{E}_v(\gamma) \neq \mathsf{m}_v(\gamma')$ by definition of rule $\mathbb{R}_{\texttt{Min}}(v)$, so in γ, we have $\alpha(\gamma, v, \mathsf{m}_v(\gamma')) = 4$ because $\mathsf{m}_v(\gamma') < \mathsf{m}_v(\gamma)$ and $\mathsf{m}_v(\gamma')$ can improve $\mathsf{m}_v(\gamma)$. Now, we have $\alpha(\gamma', v, \mathsf{m}_v(\gamma')) = 3$ because $\mathsf{E}_v(\gamma') \neq \mathsf{m}_v(\gamma')$. $\mathsf{m}_v(\gamma') < \mathsf{m}_v(\gamma)$, so if the scheduler activates only v for rule $\mathbb{R}_{\texttt{Min}}(v)$, we obtain $\Xi(\gamma') < \Xi(\gamma)$.

To conclude, $\Xi(\gamma') < \Xi(\gamma)$ for every configurations γ and γ', when γ' occurs later than γ. □

Proof of Theorem 4. Now we prove that, in $\Gamma_{\mathcal{B}}$ if the spanning structure S contains a least one cycle $\mathtt{C}$, then at least one node v in $\mathtt{C}$ has $\texttt{ErCycle}(v) = true$. For the purpose of contradiction, let us assume the opposite. Let $\gamma \in \Gamma_{\mathcal{B}}$ and every node v in the cycle $\mathtt{C}$ in γ has $\texttt{ErCycle}(v) = false$. By definition all the nodes in $\mathtt{C}$ have a parent, and all the nodes have $\mathsf{m}_v \geqslant \mathsf{m}_{\mathsf{k}_v}$. Now, if a node v shares the same m with its parent and its child, then rule $\mathbb{R}_{\texttt{Start}}$ is enabled for v, a contradiction with $\Xi(\gamma) = 0$. In a cycle, it is not possible that all nodes have $\mathsf{m}_v > \mathsf{m}_{\mathsf{k}_v}$ (due to well foundedness of integers, at least one node v has $\mathsf{m}_v < \mathsf{m}_{\mathsf{k}_v}$), which is a contradiction with the assumption that every node v is such that $\texttt{ErCycle}(v) = false$. □

Lemma 7. *Algorithm* **Break** *converges in* $O(n^n)$ *steps.*

Proof. Direct by the potential function $\Xi(\gamma)$. □

3.4.3 Silent self-stabilizing cycle detection with compact identifiers

We refine algorithm **Break** to make use of compact identifiers (of size $O(\log \log n)$ instead of global identifiers (of size $O(\log n)$). With compact identifiers, the main problem is the following: two nodes u and v can deduce that $\mathsf{Cid}_u =_c \mathsf{Cid}_u$ if and only if they have observed $\mathsf{Cid}_u^i \simeq_c \mathsf{Cid}_v^i$ during every phase i, with $1 \leqslant i \leqslant \widehat{\mathsf{B}}_v$. A node v selects the minimum compact identifier stored in variable m in its neighborhood (including itself). If it is the case that in a previous configuration, one of v's children presented v a compact identifier smaller than its own, v became passive (Variable $\mathsf{Active}_v = false$), and remained active otherwise (Variable $\mathsf{Active}_v = true$). Only active nodes can continue to increase their phase. Moreover, a node increases its phase if and only if its parent and one of its children u has the same information, namely $\mathsf{Cid}_u^i \simeq_c \mathsf{Cid}_v^i \simeq_c \mathsf{Cid}_{\mathsf{p}_v}^i$. Observe that in a spanning tree, it is possible that several nodes do not increase their phases. For example, leaf nodes are in such a situation. To explain why the absence of phase increases does not cause trouble, let v be the node with the smallest identifier involved in a cycle and let us suppose that v has two children, one child u involved in the cycle, and another child w that isn't. In some configuration, w may not be able to increase its phase, but then u reaches the same phase of the active node v, so v increases its phase, and the system reaches a configuration where $\mathsf{Cid}_u^i =_c \mathsf{Cid}_v^i$. Then, v detects an cycle error. Variable E_v combined to this compact identifier usage permits to obtain a silent algorithm.

Predicate $\texttt{ErCycle}$ now takes into account the error(s) related to compact identifiers management. It is important to note that the cycle breaking algorithm does not manage phase differences. Indeed, a node v whose phase is bigger than that of one of its children u assigns m_u to m_v, if and only if $\mathsf{ph}_v > \mathsf{ph}_u + 1$ or no child of v has the same compact identifier as v. The m_v variable is be compared using lexicographic order by rule $\mathbb{R}_{\texttt{Min}}$. The modifications to algorithm **Break** are minor. We add only one rule to increases the phase: $\mathbb{R}_{\texttt{Inc}}$. Only a passive node can restart. Remark that now the m variable uses $O(\log \log n)$ bits. As the p variables store a color, we obtain a memory requirement of $O(\max\{\log \Delta, \log \log n\})$ bits per node.

3.4.4 Algorithm C-Break and Predicates

Predicate `ErCycle` must be updated to take into account this extra care. We denote by k_v the child of v with minimum compact identifier stored in m_v. Moreover, predicate `ErCycle` now takes into account the error(s) related to compact identifiers management (see Equation 1: $\texttt{ErT}(v)$). It is important to note that the cycle breaking algorithm does not manage phase differences. The compact identifier stored in m_v is be compared using lexicographic order by rule $\mathbb{R}_{\texttt{Min}}$.

$$\texttt{ErCycle}(v) \equiv (\mathsf{p}_v \neq \varnothing) \wedge \Big((\mathsf{m}_{\mathsf{k}(v)} =_c \mathsf{Cid}_v^i) \vee (\exists (u,w) \in \texttt{Ch}(v) : \mathsf{m}_u =_c \mathsf{m}_w) \vee (\mathsf{m}_v >_c \mathsf{Cid}_v^i) \vee \big((\mathsf{m}_v \not\simeq_c \mathsf{Cid}_v) \wedge (\mathsf{m}_v <_c \mathsf{m}_{\mathsf{k}(v)})\big) \vee (\mathsf{Active}_v \wedge \texttt{ErT}(v))\Big) \tag{31}$$

Algorithm 4: Algorithm **C-Break** For node v with $\neg\texttt{ErCycle}(v)$

$$\begin{array}{lll}
\mathbb{R}_{\texttt{Inc}} & : \mathsf{Active}_v \wedge (\mathsf{m}_{\mathsf{p}_v} \simeq_c \mathsf{m}_v \simeq_c \mathsf{m}_{\mathsf{k}(v)}) & \longrightarrow \mathcal{I}ncPh(v); \\
\mathbb{R}_{\texttt{Start}} & : \neg\mathsf{Active}_v \wedge (\mathsf{m}_{\mathsf{p}_v} \simeq_c \mathsf{m}_v \simeq_c \mathsf{m}_{\mathsf{k}(v)}) \wedge (\mathsf{E}_v \not\simeq_c \mathsf{m}_v) & \longrightarrow \mathsf{E}_v := \mathsf{m}_v; \\
\mathbb{R}_{\texttt{Min}} & : (\mathsf{m}_v >_c \mathsf{m}_{\mathsf{k}(v)}) \wedge (\mathsf{E}_v \not\simeq_c \mathsf{m}_{\mathsf{k}(v)}) & \longrightarrow \mathsf{m}_v := \mathsf{m}_{\mathsf{k}(v)}, \mathsf{Active}_v := \mathit{false}; \\
\mathbb{R}_{\texttt{ID}} & : (\mathsf{E}_{\mathsf{p}_v} \simeq_c \mathsf{E}_v \simeq_c \mathsf{E}_{\mathsf{k}(v)} \simeq_c \mathsf{m}_v) \wedge (\mathsf{m}_v \not\simeq_c \mathsf{Cid}_v^1) & \longrightarrow \mathsf{m}_v := \mathsf{Cid}_v^1, \mathsf{Active}_v := \mathit{true};
\end{array}$$

Theorem 5. *The algorithm* **C-Break** *solves the detection of cycle in arbitrary n-node graph in a silent self-stabilizing manner, assuming the state model, and a distributed unfair scheduler. Moreover, if the n node identifiers are in $[1, n^c]$, for some $c \geqslant 1$, then algorithm* **C-Break** *uses $O(\max\{\log\Delta, \log\log n\})$ bits of memory per node and converges in $O(n^n \log n)$ steps.*

The proof of theorem 5 mimics the proof of algorithm **Break**. The extra $\log n$ steps factor (with respect to algorithm **Break**) results from the number of comparisons that are necessary when using compact identifiers.

4 Talkative spanning tree-construction without distance to the root

Our approach for self-stabilizing leader election is to construct a spanning tree whose root is to be the elected leader. Two main obstacles to self-stabilizing tree-construction are the possibility of an arbitrary initial configuration containing one or more cycles, or the presence of one or more impostor-rooted spanning trees. We already explained how the cycle detection and cleaning process takes place, so we focus in this section on *cycleless* configurations.

The main idea is to mimics the *fragments* approach introduced by Gallager *et al.* [29]. In an ideal situation, at the beginning each node is a fragment, each fragment merges with a neighbor fragment holding a bigger root signature, and at the end remains only one fragment, rooted in the root with the biggest signature (that is, the root with maximum degree, maximum color, and maximum global identifier). To maintain a spanning structure, the neighbors that become relatives (that is, parents or children) remain relatives after that. Note that the relationship may evolve through time (that is, a parent can become a child and vice versa). So our algorithm maintains that as an invariant (see Lemma 8).

Indeed, when two fragments merge, the one with the root with smaller signature F_1 and the other one with a root with bigger signature F_2, the root of F_1 is re-rooted toward its descendants until reaching the node that identified F_2. This approach permits to construct an acyclic spanning structure, without having to maintain distance information. The variable R_v stores the signature relative to the root (that is, its degree, its color, and its identifier). Note that, the comparison between two R is done using lexical ordering. The variable new_v stores the color of the neighbor w of v leading to u with $\mathsf{R}_u > \mathsf{R}_v$ if there exists such a node, and $\varnothing$ otherwise. The function $\mathtt{f}(v)$ returns the color of the neighbor of v with the maximum root (see 32).

4.1 Algorithm description

Let us now give more details about our algorithm (presented in Algorithm 5). If a root v has a neighbor u with $\mathsf{R}_u > \mathsf{R}_v$, then v chooses u as its parent (see rule $\mathbb{R}_{\mathtt{Merge}}$). If a node v (not a root) has a neighbor u with $\mathsf{R}_u > \mathsf{R}_v$, it stores its neighbor's color in Variable new_v, and updates its R_v to R_u. Yet, it does not change its parent. This behavior creates a path (thanks to Variable new) between a root r of a sub spanning tree $\mathtt{T}_r$ and a node contained in an other sub spanning tree $\mathtt{T}_{r'}$ rooted in r', with $\mathsf{R}_{r'} > \mathsf{R}_r$ (see rule $\mathbb{R}_{\mathtt{Path}}$). The subtree $\mathtt{T}_r$ is then re-rooted toward a node aware of a root with a bigger signature u. Now, when $v \in \mathtt{T}_r$'s neighbor u becomes root, it takes u as a parent (see rules $\mathbb{R}_{\mathtt{ReRoot}}$ and $\mathbb{R}_{\mathtt{Del}}$). Finally, the descendants of the re-rooted root update their root variables (see rule $\mathbb{R}_{\mathtt{Update}}$). The predicate $\mathtt{ErST}(v)$ (see 33) captures trivial

Algorithm 5: Algorithm **ST**

$\mathbb{R}_{\mathtt{Del}}$: $(\mathsf{p}_v \neq \varnothing) \wedge (\mathsf{p}_{\mathsf{p}_v} = \mathsf{c}_v) \longrightarrow \mathsf{p}_v := \varnothing;$

$\mathbb{R}_{\mathtt{Update}}$: $(\mathsf{p}_v = \mathtt{f}(v)) \wedge (\mathsf{p}_{\mathsf{p}_v} \neq \mathsf{c}_v) \wedge (\mathtt{f}(v) \neq \varnothing) \wedge (\mathsf{R}_v < \mathsf{R}_{\mathtt{f}(v)}) \wedge (\mathsf{new}_v = \varnothing) \longrightarrow \mathsf{R}_v := \mathsf{R}_{\mathtt{f}};$

$\mathbb{R}_{\mathtt{Path}}$: $(\mathsf{p}_v \notin \{\varnothing, \mathtt{f}(v)\}) \wedge (\mathsf{p}_{\mathsf{p}_v} \neq \mathsf{c}_v) \wedge (\mathtt{f}(v) \neq \varnothing) \wedge (\mathsf{R}_v < \mathsf{R}_{\mathtt{f}(v)}) \wedge (\mathsf{new}_v = \varnothing)$
$\longrightarrow (\mathsf{R}_v, \mathsf{new}_v) := (\mathsf{R}_{\mathtt{f}(v)}, \mathtt{f}(v));$

$\mathbb{R}_{\mathtt{Merge}}$: $(\mathsf{p}_v = \varnothing) \wedge (\mathtt{f}(v) \neq \varnothing) \wedge (\mathsf{R}_v < \mathsf{R}_{\mathtt{f}(v)}) \wedge (\mathsf{new}_{\mathtt{f}(v)} = \varnothing) \longrightarrow (\mathsf{p}_v, \mathsf{R}_v) := (\mathtt{f}(v), \mathsf{R}_{\mathtt{f}});$

$\mathbb{R}_{\mathtt{ReRoot}}$: $(\mathsf{p}_v = \varnothing) \wedge (\mathtt{f}(v) \neq \varnothing) \wedge (\mathsf{R}_v = \mathsf{R}_{\mathtt{f}(v)}) \wedge (\mathsf{new}_v \neq \varnothing) \longrightarrow (\mathsf{p}_v, \mathsf{new}_v) := (\mathsf{new}_v, \varnothing);$

errors and impostor-root errors for the construction of the spanning tree. Note that it is used in **Freeze** only (and not in **ST**) as these errors are never created by **ST** and **Freeze** has higher priority than **Freeze** (see Section 4.2).

4.1.1 Predicates

The function $\mathtt{f}(v)$ returns the color of the neighbor of v with the maximum root:

$$\mathtt{f}(v) = \{\mathsf{c}_u : u \in N(v) \wedge \mathsf{R}_u = \max\{\mathsf{R}_w : w \in N(v)\}\} \tag{32}$$

We now present a list of trivial errors and impostor-root errors for the construction of the spanning tree. The explanations of the different elements composing the predicate $\mathtt{ErST}(v)$ follow: (1) A node without relative has its root signature different to its own variables. (2) The variable δ_v is not equal to the degree of v. (3) The invariant is not satisfied. (4) A node with $\mathsf{new}_v = \varnothing$ (that is, v is not involved in a rerouting process) has a root signature bigger than that of its parent;(5) A node with $\mathsf{new}_v \neq \varnothing$ (that is, v is involved in a rerouting process) has a root signature different from that of its tentative new parent.(6) A root v with $\mathsf{new}_v = \varnothing$ and a signature R_v that does not match is own.(7)A node involved in a rerouting process whose parent's parent is itself.

$$\texttt{ErST}(v) \equiv \begin{cases} (1) & \big((\mathsf{p}_v = \varnothing) \wedge (\texttt{Ch}(v) = \varnothing) \wedge [(\mathsf{R}_v \neq (\deg_v, \mathsf{c}_v, \text{ID}_v)) \vee (\mathsf{new}_v \neq \varnothing)]\big) \vee \\ (2) & (\delta_v \neq \deg_v) \vee \\ (3) & (\overline{\mathsf{n}}(v) \notin \{\{\mathsf{p}_v\} \cup \texttt{Ch}(v)\}) \vee \\ (4) & \big((\mathsf{new}_v = \varnothing) \wedge (\mathsf{R}_v > \mathsf{R}_{\mathsf{p}_v})\big) \vee \\ (5) & \big((\mathsf{new}_v \neq \varnothing) \wedge (\mathsf{R}_v \neq \mathsf{R}_{\mathsf{new}_v})\big) \vee \\ (6) & \big((\mathsf{p}_v = \varnothing) \wedge (\mathsf{new}_v = \varnothing) \wedge (\mathsf{R}_v \neq (\deg_v, \mathsf{c}_v, \text{ID}_v))\big) \vee \\ (7) & \big((\mathsf{new}_v \neq \varnothing) \wedge (\mathsf{p}_{\mathsf{p}_v} = \mathsf{c}_v)\big) \end{cases} \tag{33}$$

Examples of the possible situations are presented in Figure 6.

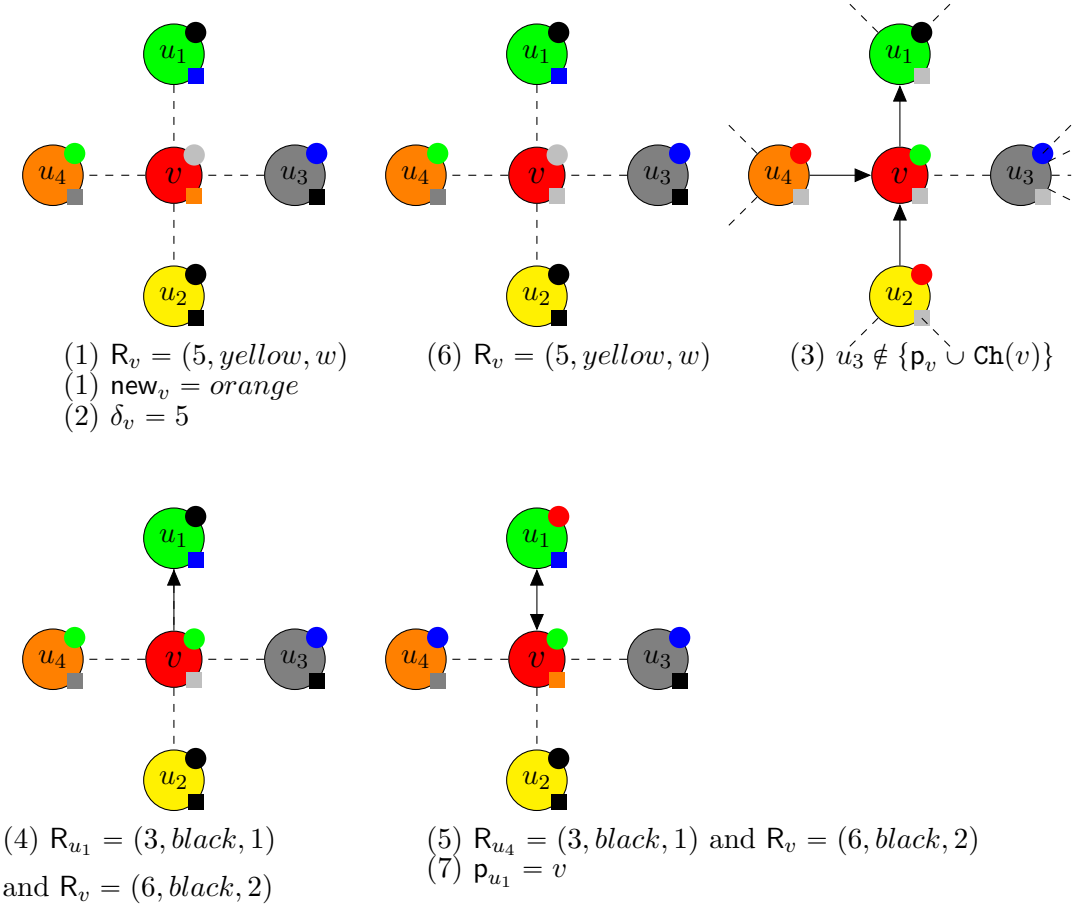

Figure 6: We present various situations arising when evaluating predicate $\texttt{ErST}(v)$ to true. The node color denotes its c_v variable, the top right circle denote its p_v variable (light gray represents no color), the bottom right square denotes new_v variable (light gray represents no color), and the arrow visually represent the parent-child relationship. Under each situation, the multiset of numbers represents which conditions of predicate $\texttt{ErST}(v)$ evaluate too true. For example, in the first situation, v has degree four yet announces a degree of 5, hence condition (2) of predicate $\texttt{ErST}(v)$ evaluates to true.

4.2 Correctness

Theorem 6. *Algorithm* **ST** *solves the spanning tree-construction problem in a silent self-stabilizing way in any n-node graph, assuming the absence of spanning cycle, the state model, and a distributed unfair scheduler, using* $O(\log n)$ *bits of memory per node.*

Proof of Theorem 6 Let $\overline{\mathsf{n}}(v)$ denote the color of v's neighbor with the maximum degree, and in case there are several such neighbors, the one with maximum color.

$$\overline{\mathsf{n}}(v) = \{\mathsf{c}_u : u \in N(v) \wedge \delta_u = \max\{\delta_w : w \in N(v)\} \wedge \mathsf{c}_u = \max\{\mathsf{c}_w : w \in N \wedge (\delta_w = \delta_u)\}\} \quad (34)$$

Lemma 8 (Invariant)**.** *For every node* $v \in V$ *such that* $\mathsf{p}_v \neq \varnothing$ *and* $\mathtt{Ch}(v) \neq \varnothing$, $\overline{\mathsf{n}}(v) \in \{\mathsf{p}_v \cup \mathtt{Ch}(v)\} \neq \varnothing$ *remains true.*

Proof. Proof by induction

Basis case: When a node v starts the algorithm it chooses for a parent the node with the maximum degree, if there exist more than one it chooses the one with the maximum identifier among the ones with the maximum degree. So if a node picks a parent u at the first execution of the algorithm it takes $u = \overline{\mathsf{n}}(v)$ so for theses nodes the invariant is preserved. For the nodes v which are a maximum local. We denote by u the node $\overline{\mathsf{n}}(v)$. Suppose that at the first execution of u, u choses the node w as a parent, that means $\mathsf{R}_w > \mathsf{R}_u$ and $\mathsf{R}_w > \mathsf{R}_v$. So after this execution $\mathsf{R}_u = \mathsf{R}_w$, so now v can choose u as a parent and the invariant is preserved for node v. So after one execution of the algorithm for all the nodes the invariant is preserved.

Assumption: Assume true that after t steps of execution, the algorithm preserved the invariant.

Inductive step: Let us consider a node v, by the assumption we have

$$\overline{\mathsf{n}}(v) \cap \{\{\mathsf{p}_v\} \cup \mathtt{Ch}(v)\} \neq \varnothing$$

The node v cannot change its children in can only change its parent, and only if v its a root (see rules $\mathbb{R}_{\mathtt{Merge}}$ and $\mathbb{R}_{\mathtt{ReRoot}}$ of Algorithm 5). So for v $\overline{\mathsf{n}}(v) \in \mathtt{Ch}(v)$, the rule $\mathbb{R}_{\mathtt{Merge}}$ assigns as a parent a new neighbor u ($u \notin \mathtt{Ch}(v)$) so the invariant is preserved. The rule $\mathbb{R}_{\mathtt{ReRoot}}$ assigns as a parent of v a child of v so the invariant is preserved.

□

Lemma 9. *The descendants* u *of* v *with* $\mathsf{new}_u = \varnothing$ *have* $\mathsf{R}_u \leqslant \mathsf{R}_v$.

Proof. Proof by induction on the value R_u with u descendants of v

Base case: Each node $v \in V$ with $\mathsf{p}_v = \varnothing$ and $\mathtt{Ch}(v) = \varnothing$ has $\mathsf{R}_v = (\deg_v, \mathsf{c}_v, \mathrm{ID}_v)$ and $\mathsf{new}_v = \varnothing$, otherwise an error is detected. A node v takes a parent iff there exists a neighbor w of v such that $\mathsf{R}_w > \mathsf{R}_v$, and in this case v maintains its variable $\mathsf{new}_w = \varnothing$, so the claim is satisfied (see Rule $\mathbb{R}_{\mathtt{Merge}}$).

Assumption: Assume that there exists a configuration γ where for every node $v \in V$, all the descendants u of v with $\mathsf{new}_u = \varnothing$ have $\mathsf{R}_u \leqslant \mathsf{R}_v$.

Inductive step: We consider Configuration $\gamma + 1$. For a node v and every descendants u, the assumption gives the property that if $\mathsf{new}_u = \varnothing$, then $\mathsf{R}_u \leqslant \mathsf{R}_v$. Let us now consider the case where there exists a neighbor w of u with $\mathsf{R}_u < \mathsf{R}_w$.

– If w is the parent of u, R_u takes the value of R_w (see rule $\mathbb{R}_{\texttt{Update}}$). By the induction assumption, we have $\mathsf{R}_w \leqslant \mathsf{R}_v$ (as a parent of u, w is also a descendant of v). So, R_u remains inferior or equal to R_v.
– By the induction assumption, if $\mathsf{R}_w > \mathsf{R}_u$, then w cannot be a descendant of u.
– If w is not in the same subtree of u, u cannot change its parent because u is not a root (see rule $\mathbb{R}_{\texttt{Merge}}$). So u changes its R_u to R_w, but it sets $\mathsf{new}_u = w$ (see rule $\mathbb{R}_{\texttt{Path}}$).

Now, if there exists a neighbor w of u such that $\mathsf{R}_w = \mathsf{R}_u$, then to execute rule $\mathbb{R}_{\texttt{ReRoot}}$, u must be a root. We obtain a contradiction with our assumption that u is a descendant of v.

To conclude, if u is the descendant of v in configuration γ and it remains a descendant of v at configuration $\gamma+1$, and the value of new_u remains empty, then $\mathsf{R}_u \leqslant \mathsf{R}_v$ in configuration $\gamma + 1$.

□

Lemma 10. *If there exists an acyclic spanning structure in Configuration γ, then any execution of a rule maintains an acyclic spanning structure in Configuration $\gamma + 1$.* [2]

Proof. Proof by induction on the size of the acyclique spanning structure.

Basis case: By contradiction: Remark that, thanks to Algorithm **C-Color**, there exist a total order between the neighbors of a node. Let us consider three neighbor nodes $a, b, c \in V$ such that in Configuration γ, a, b and c have no relatives. Then all three nodes are enabled by rule $\mathbb{R}_{\texttt{Merge}}$. Let us suppose for the purpose of contradiction that in Configuration $\gamma+1$ a cycle exists. More precisely: $\mathsf{p}_a = b$, $\mathsf{p}_b = c$ and $\mathsf{p}_c = a$, to achieve that :

1. a must choose b as a parent, for that $\mathsf{R}_b > \mathsf{R}_c$
2. b must choose c as a parent, for that $\mathsf{R}_c > \mathsf{R}_a$
3. c must choose a as a parent, for that $\mathsf{R}_a > \mathsf{R}_b$

We obtain a contradiction between (1),(2) and (3).

Assumption: Assume true that in configuration γ there exists an acyclic spanning structure.

Inductive step: Let us consider a node $v \in V$, a node v takes a new parent only in two cases, and in both case, v must be a root.

Let us consider first rule $\mathbb{R}_{\texttt{Merge}}$, let u be the neighbor of v with $\mathsf{R}_v < \mathsf{R}_u$ and $\mathsf{new}_u = \varnothing$. By Lemma 9, u is not a descendant of v, so if v takes u as a new parent, an acyclic spanning structure is preserved. Now, we consider rule $\mathbb{R}_{\texttt{ReRoot}}$. Let u be the neighbor of v such that $\mathsf{R}_u = \mathsf{R}_v$ and $\mathsf{new}_v = u$. In this case, u is either a child of v with $\mathsf{new}_u \neq \varnothing$, or v is not a child of v with $\mathsf{new}_u = \varnothing$. If v is a child of v, v takes u as a parent. Remark that the first action of u is to delete its parent (see rule $\mathbb{R}_{\texttt{Del}}$, and consider the fact that all other rules require $\mathsf{p}_v \notin \texttt{Ch}(v)$), so we do not consider this case as a cycle. If v is not a child of v with $\mathsf{new}_u = \varnothing$, by Lemma 9 u is not a descendant of v. Now, when v takes u as a parent, this action maintains an acyclic spanning structure.

To conclude, Configuration $\gamma + 1$ maintains a acyclic spanning structure.

□

[2] When a node v has $\mathsf{p}_v \in \texttt{Ch}(v)$, we delete p_v (see rule $\mathbb{R}_{\texttt{Del}}$), so we do not consider this case as a cycle.

Lemma 11. *If v is a node such that $\mathsf{R}_v = \mathsf{R}_r$, and every ancestor of v (and v itself) have $\mathsf{new} = \varnothing$. Then r is an ancestor of v, or v itself.*

Proof. Suppose for the purpose of contradiction that $r \neq v$, and r is not an ancestor of v. By Lemma 10, v is an element of a sub spanning tree $\mathtt{T}$. Let w be the oldest ancestor of v such that $\mathsf{new}_w = \varnothing$. By hypothesis, every ancestor z of v (including w) has $\mathsf{R}_z \neq \mathsf{R}_r$. By Lemma 9, we have $\mathsf{R}_v \leqslant \mathsf{R}_z$, which contradicts $\mathsf{R}_v = \mathsf{R}_r$. □

Lemma 12. *Executing Algorithm* **ST** *constructs a spanning tree rooted in the node with the maximum degree, maximum color, maximum identifier, assuming the state model, and a distributed unfair scheduler.*

Proof. Let $\psi : \Gamma \times V \to \mathbb{N}$ be the function defined by:

$$\psi(\gamma, v) = \big((\deg_\ell - \delta_v) + (\mathsf{c}_\ell - \mathsf{c}_v) + (\mathrm{ID}_\ell - \mathrm{ID}_v)\big)$$

where ℓ is the node with $\mathsf{R}_\ell > \mathsf{R}_v$ with $v \in V \backslash \{\ell\}$. Now, let $\phi : \Gamma \times V \to \mathbb{N}$ be the function defined by:

$$\phi(\gamma, v) = \begin{cases} 2 & \text{if } \mathsf{new}_v \neq \varnothing \\ 1 & \text{if } \mathsf{p}_{\mathsf{p}_v} = \mathsf{c}_v \\ 0 & \text{otherwise} \end{cases}$$

Remark that a node v cannot have $\mathsf{new}_v \neq \varnothing$ and $\mathsf{p}_v \in \mathtt{Ch}(v)$. Otherwise an error is detected through predicate $\mathtt{ErST}(v)$.

Let $\Psi : \Gamma \to \mathbb{N}^2$ be the potential function defined by:

$$\Psi(\gamma) = \Big(\sum_{v \in V} \psi(\gamma, v), \sum_{v \in V} \phi(\gamma, v) \Big).$$

Let γ be a configuration such that $\Psi(\gamma) > 0$, and let v be a node in V such that v is enabled by a rule of Algorithm **ST**. If v executes rules $\mathbb{R}_{\mathtt{Update}}$, $\mathbb{R}_{\mathtt{Merge}}$, or $\mathbb{R}_{\mathtt{Path}}$, then R_v increases and we obtain $\psi(\gamma', v) < \psi(\gamma, v)$. Now, if v executes rule $\mathbb{R}_{\mathtt{ReRoot}}$, this implies new_v is not empty. After execution of $\mathbb{R}_{\mathtt{ReRoot}}$, new_v become empty, so $\phi(\gamma, v)$ decreases by one. Finally, if v executes rule $\mathbb{R}_{\mathtt{Del}}$, it implies that v had $\mathsf{p}_{\mathsf{p}_v} = \mathsf{c}_v$, and now $\mathsf{p}_v = \varnothing$. As a result, $\phi(\gamma', v) = \phi(\gamma, v) - 1 = 0$.

Therefore, we obtain $\Psi(\gamma') < \Psi(\gamma)$. By Lemmas 10 and 11 we obtain the property that when $\Psi(\gamma) = 0$, a spanning tree rooted in ℓ is constructed. □

Lemma 13. *Algorithm* **ST** *converges in $O(\Delta n^3)$ steps.*

Proof. Direct by the potential function $\Psi(\gamma)$. □

We adapt **ST** to use compact identifiers and obtain Algorithm **C-ST**. It is simple to compare two compact identifiers when the nodes are neighbors. Along the algorithm execution, some nodes become non-root, and therefore the remaining root of fragments can be far away, separated by non-root nodes. To enable multi-hop comparison, we use a broadcasting and convergecast wave on a spanning structure to assure the propagation of the compact identifier.

Theorem 7. C-ST *solves the spanning tree-construction problem in a talkative self-stabilizing way in any n-node graph, assuming the absence of spanning cycle, the state model, and a distributed unfair scheduler, in $O(\log \Delta + \log \log n)$ bits of memory per node.*

Let v a node that wants to broadcast its compact identifier. We add a variable check to our previous algorithm. This variable checks whether every descendant or neighbor shares the same compact identifier at the same phase before proceeding to the convergecast. More precisely, a node u must checks if every neighbors w has $\mathsf{Cid}_u \simeq_c \mathsf{Cid}_w$, and if every child has $\mathsf{check}_v = true$. If so, it sets its variable $\mathsf{check}_v = true$, and the process goes on until node v. As a consequence, v increases or restarts its phase and assigns $false$ to check.

Lemma 14. *Algorithm* **C-ST** *converges in* $O(\Delta n^3 \log n)$ *steps.*

The proof of Theorem 7 mimics the proof of Theorem 6.

5 Self-stabilizing leader election

We now present the final assembly of tools we developed to obtain a self-stabilizing leader election algorithm. We add to Algorithm **C-ST** an extra variable ℓ that is maintained as follows: if a node v has no parent, then $\ell_v = true$, otherwise, $\ell_v = false$. Variable ℓ_v is meant to be the output of the leader election process.

Our self-stabilizing leader election algorithm results from combining severals algorithms. As already explained (see Figure 1), a higher priority algorithm resets all the variables used by lesser priority algorithms. Moreover, lesser priority algorithm do not modify the variables of the higher priority algorithms. Algorithms are prioritized as follows: **C-Color**, **Freeze**, **C-Break** and **C-ST**. First, starting from an arbitrary configuration, **C-Color** eventually guaranteed that colors form a distance two coloring in the network, and those colors never change thereafter (the algorithm is silent). Then, **C-Break** ensure that no cycles or fake spanning tree go undetected forever. If one is found, **Freeze** is used (with a higher priority) to destroy it. So, after **C-Break** terminates (and it does since it is silent), no cycle of fake spanning tree exists.

Only algorithm **C-ST** is talkative, but the number of steps before electing a single leader forever is bounded once Algorithms **C-Color**, **Freeze**, and **C-Break** all terminate. Thanks to Theorem 7, we obtain in a finite number of steps a spanning tree rooted in the node with the maximum degree, maximum color, and maximum identifier (in lexicographic order). Adding a leader variable as suggested in this section to Algorithm **C-ST** guarantees that only the root of the spanning tree r has $\ell_r = true$ and every other node $v \in V \backslash \{r\}$ has $\ell_v = false$. Since the root of the spanning tree remains the same forever, so does the elected leader.

Theorem 1. *Algorithm called* **C-LE** *solves the leader election problem in a talkative self-stabilizing manner in any n-node graph, assuming the state model and a distributed unfair scheduler, with* $O(\log \Delta + \log \log n)$ *bits of memory per node.*

Proof of Theorem 1. We first need to show that the number of activations of rules of algorithm **C-ST** are bounded if there exist nodes enabled by **C-Color**, **Freeze** or **C-Break**. Let us consider a subset of the nodes A enabled for at least one of these algorithms, and by S the nodes enabled by rule **C-ST**. The nodes in S belong to some spanning trees (possibly only one), otherwise at least one of rules of **Freeze** or **C-Break** would be enabled. So, there exist a node in S that is enabled by algorithm **C-ST**. Algorithm **C-ST** is talkative, but it runs by waves, and its waves require that all neighbors of a node v have the same R at each phase. As we consider connected graphs only, there exists at least one node v in S with a neighbor u in A. Then, there exists a configuration γ' where the rules of **C-ST** are not enabled, because u cannot have the same R at each phase (since u is not enabled by rules of **C-ST**). So, only Algorithms **C-Color**, **Freeze**, and **C-Break** may now be scheduled for execution, as they have

higher priority. As they are silent and operate under an unfair distributed scheduler, we obtain convergence.

Let us now consider a configuration γ where no node are enabled for Algorithm **C-Color**, **Freeze**, and **C-Break**. There exists a node enabled by Algorithm **C-ST**. Thanks to Theorem 7, we obtain a spanning tree rooted at the node with the maximum degree, maximum color, and maximum identifier. As a consequence, only the root r has $\ell_r = true$ and every other node $v \in V \backslash \{r\}$ has $\ell_v = false$. □

6 Conclusion

We presented the first self-stabilizing leader election for arbitrary graphs of size n that uses $o(\log n)$ bits of memory per node, breaking a long-standing lower bound. Our solution does not require any weakening of the usual self-stabilization model, in partucular it withstands the most general scheduling assumption: the *unfair* scheduler. Besides tree construction and leader election, our research paves the way for new memory efficient self-stabilizing algorithms. For example, some of the solutions for self-stabilizing maximal matching construction use a fixed number of "pointer to neighbor" variables [33]. Using our distance two coloring process would permit to go from $O(\log n)$ to $O(max\{\log \Delta, \log \log n\})$ bits of memory per node.

In the case of ring shaped networks, an important byproduct of our approach is that, with respect to previous work [14], we no longer require the hypothesis of weak fairness (simple progress is sufficient), while the space complexity is not altered. Indeed, in a ring, $\Delta = 2$, so our space complexity becomes $O(\log \log n)$ bits per nodes, which is the same as in previous work [14].

Although there exists several techniques and methods to prove self-stabilization is a systematic manner [27, 28, 20, 26, 41], in the context of self-stabilization they are currently limited to systems that reach a fixed point (*a.k.a.* a fixed single configuration) after finite time. This implies they may only be used for *silent* protocols. Alas, silent solutions to leader election require $\Omega(\log n)$ bits per node [23]. Instead, we developped a systematic approach based on potential functions that allow to obtain both correctness proofs and step complexity results. We plan to further formalize these techniques in future work.

References

[1] J. Adamek, M. Nesterenko, and S. Tixeuil. Using abstract simulation for performance evaluation of stabilizing algorithms: The case of propagation of information with feedback. In SSS 2012, LNCS. Springer, 2012.

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

[3] Y. Afek, S. Kutten, and M. Yung. Memory-efficient self stabilizing protocols for general networks. In WDAG '90, pages 15–28, 1990.

[4] A. Arora and M. G. Gouda. Distributed reset. IEEE Trans. Computers, 43(9):1026–1038, 1994.

[5] Anish Arora, Paul Attie, Michael Evangelist, and Mohamed Gouda. Convergence of iteration systems. In J. C. M. Baeten and J. W. Klop, editors, CONCUR '90 Theories of Concurrency: Unification and Extension, pages 70–82, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[6] M. Arumugam and S. S. Kulkarni. Prose: A programming tool for rapid prototyping of sensor networks. In S-CUBE, pages 158–173, 2009.

[7] B. Awerbuch and R. Ostrovsky. Memory-efficient and self-stabilizing network reset. In PODC, pages 254–263. ACM, 1994.

[8] J. Beauquier, M. Gradinariu, and C. Johnen. Memory space requirements for self-stabilizing leader election protocols. In Proceedings of PODC 1999, pages 199–208, 1999.

[9] J. R. S. Blair and F. Manne. An efficient self-stabilizing distance-2 coloring algorithm. Theor. Comput. Sci., 444:28–39, 2012.

[10] L. Blin, F. Boubekeur, and S. Dubois. A self-stabilizing memory efficient algorithm for the minimum diameter spanning tree under an omnipotent daemon. In IPDPS 2015, pages 1065–1074, 2015.

[11] L. Blin and P. Fraigniaud. Space-optimal time-efficient silent self-stabilizing constructions of constrained spanning trees. In Proceedings of ICDCS 2015, pages 589–598, 2015.

[12] L. Blin, M. Potop-Butucaru, and S. Rovedakis. A super-stabilizing log(n)log(n)-approximation algorithm for dynamic steiner trees. Theor. Comput. Sci., 500:90–112, 2013.

[13] Lélia Blin and Sébastien Tixeuil. Brief announcement: Compact self-stabilizing leader election in arbitrary graphs. In 31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria, pages 43:1–43:3, 2017.

[14] 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.

[15] Lélia Blin and Sébastien Tixeuil. Compact self-stabilizing leader election for general networks. In LATIN 2018: Theoretical Informatics - 13th Latin American Symposium, Buenos Aires, Argentina, April 16-19, 2018, Proceedings, pages 161–173, 2018.

[16] N.S. Chen, H.P. Yu, and S.T. Huang. A self-stabilizing algorithm for constructing spanning trees. Information Processing Letters, 39(3):147 – 151, 1991.

[17] Y. Choi and M. G. Gouda. A state-based model of sensor protocols. Theor. Comput. Sci., 458:61–75, 2012.

[18] Z. Collin and S. Dolev. Self-stabilizing depth-first search. Information Processing Letters, 49(6):297 – 301, 1994.

[19] A. R. Dalton, W. P. McCartney, K. Ghosh Dastidar, J. O. Hallstrom, N. Sridhar, T. Herman, W. Leal, A. Arora, and M. G. Gouda. Desal alpha: An implementation of the dynamic embedded sensor-actuator language. In ICCCN, pages 541–547. IEEE, 2008.

[20] S. Delaët, B. Ducourthial, and S. Tixeuil. Self-stabilization with r-operators revisited. Journal of Aerospace Computing, Information, and Communication (JACIC), 3(10):498–514, 2006.

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

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

[23] S. Dolev, M. G. Gouda, and M. Schneider. Memory requirements for silent stabilization. Acta Inf., 36(6):447–462, 1999.

[24] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. Distributed Computing, 7(1):3–16, 1993.

[25] S. Dubois and S. Tixeuil. A taxonomy of daemons in self-stabilization. Technical Report 1110.0334, ArXiv eprint, October 2011.

[26] B. Ducourthial. r-semi-groups: A generic approach for designing stabilizing silent tasks. In Toshimitsu Masuzawa and S. Tixeuil, editors, SSS, pages 281–295. Univ. Paris 6, 2007.

[27] B. Ducourthial and S. Tixeuil. Self-stabilization with r-operators. Distributed Computing (DC), 14(3):147–162, July 2001. URL: http://www.springerlink.com/content/0p4g0yt8vkd9jnlp/, doi:10.1007/PL00008934.

[28] B. Ducourthial and S. Tixeuil. Self-stabilization with path algebra. Theoretical Computer Science (TCS), 293(1):219–236, February 2003.

[29] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. ACM Trans. Program. Lang. Syst., 5(1):66–77, 1983.

[30] M. Gradinariu and C. Johnen. Self-stabilizing neighborhood unique naming under unfair scheduler. In Proceedings of Euro-Par 2001, pages 458–465, 2001.

[31] T. Herman and S. V. Pemmaraju. Error-detecting codes and fault-containing self-stabilization. Inf. Process. Lett., 73(1-2):41–46, 2000.

[32] T. Herman and S. Tixeuil. A distributed tdma slot assignment algorithm for wireless sensor networks. In Proceedings of AlgoSensors 2004, number 3121, pages 45–58, 2004.

[33] M. Inoue, F.o Ooshita, and S. Tixeuil. An efficient silent self-stabilizing 1-maximal matching algorithm under distributed daemon without global identifiers. In Proceedings of SSS 2016., pages 195–212, 2016.

[34] G. Itkis and L. A. Levin. Fast and lean self-stabilizing asynchronous protocols. In FOCS, pages 226–239. IEEE Computer Society, 1994.

[35] G. Itkis, C. Lin, and J. Simon. Deterministic, constant space, self-stabilizing leader election on uniform rings. In WDAG, LNCS, pages 288–302. Springer, 1995.

[36] A. Korman, S. Kutten, and T. Masuzawa. Fast and compact self stabilizing verification, computation, and fault detection of an mst. In Proceedings of PODC 2011, 2011.

[37] A. J. Mayer, Y. Ofek, R.l Ostrovsky, and M. Yung. Self-stabilizing symmetry breaking in constant-space (extended abstract). In STOC, pages 667–678, 1992.

[38] T. M. McGuire and M. G. Gouda. The Austin Protocol Compiler, volume 13 of Advances in Information Security. Springer, 2005.

[39] F. A. Stomp. Structured design of self-stabilizing programs. In [1993] The 2nd Israel Symposium on Theory and Computing Systems, pages 167–176, June 1993. doi:10.1109/ISTCS.1993.253472.

[40] S. Tixeuil. Algorithms and Theory of Computation Handbook, pages 26.1–26.45. CRC Press, Taylor & Francis Group, 2009.

[41] Mirko Viroli, Giorgio Audrito, Jacob Beal, Ferruccio Damiani, and Danilo Pianini. Engineering resilient collective adaptive systems by self-stabilisation. ACM Trans. Model. Comput. Simul., 28(2), March 2018. URL: https://doi.org/10.1145/3177774, doi:10.1145/3177774.