Resource efficient stabilization for local tasks despite unknown capacity links

Lélia Blin, Anaïs Durand, Sébastien Tixeuil

Resource Efficient Stabilization for Local Tasks despite Unknown Capacity Links

Lélia Blin
Université Paris Cité, CNRS, IRIF,
8 Place Aurélie Nemours, 75013 Paris, France

Anaïs Durand
Université Clermont-Auvergne,
CNRS, LIMOS, 63000 Clermont-Ferrand, France

Sébastien Tixeuil
Sorbonne Université, CNRS, LIP6, IUF
F-75005 Paris, France

June 10, 2024

Abstract

Self-stabilizing protocols enable distributed systems to recover correct behavior starting from any arbitrary configuration. In particular, when processors communicate by message passing and the communication links are unbounded, fake messages may be placed in communication links by an adversary. When the number of such fake messages is unknown, self-stabilization may require huge resources:

- generic solutions (*a.k.a.* data link protocols) require unbounded resources, which makes them unrealistic to deploy,
- specific solutions (*e.g.*, census or tree construction) require $O(n \log n)$ or $O(\Delta \log n)$ bits of memory per node, where n denotes the network size and Δ its maximum degree, which may prevent scalability.

We investigate the possibility of resource-efficient self-stabilizing protocols in this context.

Specifically, we present self-stabilizing protocols for $(\Delta + 1)$-coloring and maximal independent set construction in any n-node graph, under the asynchronous message-passing model. The problem of $(\Delta + 1)$-coloring sand maximal independent set construction are considered benchmarking problems for local tasks. Our protocols offer many desirable features. They are deterministic, converge in $O(k\Delta n^2 \log n)$ message exchanges, where k is the (unknown) initial number of (possibly corrupted) messages in a communication link. They use messages of $O(\log \log n + \log \Delta)$ bits with a memory of $O(\Delta \log \Delta + \log \log n)$ bits at each node. The resource consumption of our protocols is thus almost oblivious to the number of nodes, enabling scalability. Moreover, a striking property of our protocols is that the nodes do not need to know the number, or any bound on the number of messages initially present in each communication link of the initial (potentially corrupted) network configuration. This permits our

protocols to handle any future network with unknown message capacity communication links.

A key building block of our coloring and maximal independent set schemes is an algorithm to obtain an acyclic orientation of graph edges, that is of independent interest, and can serve as a useful tool for solving other tasks in this challenging setting.

Mots-clés: Self-stabilizing algorithm, message passing, unbounded capacity communication, nodes coloring.

1 Introduction

Self-stabilization [14, 15, 39] is a versatile technique that enables recovery after arbitrary *transient* faults hit the distributed system, where both the participating processes and the communication medium are subject to be corrupted. Roughly, a self-stabilizing protocol is able to bring the system back to a legal configuration, starting from an arbitrary initial, potentially corrupted, configuration. The core motivation for designing self-stabilizing protocols has been underlined by Varghese and Jarayam [42], who observed that, whenever processes can crash and recover, a message-passing distributed system may reach any arbitrary global state, where the local variables stored at the processes may be inconsistent, and/or the communication links may contain spurious erroneous messages. As the global state is arbitrary, one may even assume that the local variables at the nodes as well as the contents of the messages are adversarially set to prevent recovery.

It is worth pointing out that self-stabilization in the presence of fake messages in the communication links is a challenge. Specifically, it is particularly difficult to ensure recovery when no upper bound is known on the number of (possibly fake) messages in transit in the initial configuration. Conversely, if an upper bound is known, then one can reset the system to a clean configuration by, first, emptying the links, and, second, resetting all local variables using a protocol that can "trust" the messages. Hence, non-surprisingly, the vast majority of recent works in self-stabilization assumes a weaker adversary than the one we consider in this paper. In particular, a widely used model of self-stabilization is the *state model*.

1.1 State Model, and Data Link Protocols.

In the state model, processes atomically read the states of their neighboring processes for updating their state. That is, the state model abstracts away all issues related to corrupted communication media. Indeed, the state model is motivated by the fact that there exist self-stabilizing *data link* protocols [2, 11, 16, 18, 25, 32, 41]. Such protocols provide (to another protocol) a layer that ensures reliable communications between neighbors exchanging messages over unreliable communication links. Yet, the use of data link protocols such as the ones in the aforementioned previous work yields important issues.

In particular, if the initial number of spurious messages in the communication links is unknown (this also implies that the link maximum capacity is unbounded), then Gouda and Multari [25] proved that it is impossible to design a self-stabilizing data link protocol using a finite number of configurations (a.k.a.

global states). Therefore, one has to relax the constraints on the protocol. Such a relaxation may consist in designing *pseudo-stabilizing* data link protocols [11]. However, a pseudo-stabilizing protocol only guarantees that an infinite suffix of the execution satisfies the specification of the system, and hence its stabilization time becomes unbounded. Another relaxation consists of using self-stabilizing data link protocols that are *not bounded* in terms of resources. However, such protocols require either unbounded variables [18, 25, 32], unbounded code size (*a.k.a.* aperiodic functions) [2], or an ever-growing number of messages [18], which is undesirable from a practical point of view. A third relaxation consists of using *randomization* [2], but then the correctness of the system is not certain.

As a consequence, in the framework of data link self-stabilization, previous works often assume that the initial number of spurious messages present in the communication links is known to the participating processes [32], or alternatively, that the link capacity is bounded [1, 10] (hence, the initial number of spurious messages is upper bounded by the link capacity).[1] Under this assumption, very efficient self-stabilizing solutions can be obtained (see, *e.g.*, [41]). Actually, assuming that the number of erroneous messages initially present in the links is known to the nodes, even stronger self-stabilizing properties can be guaranteed, such as *snap-stabilization* [16] (a snap-stabilizing data link protocol guarantees that reliable communications between participating processes are immediately available after a failure).

From the above, one can conclude that, while the knowledge (or approximate knowledge) of the number of possible initial spurious messages in the communication links enables the design of efficient self-stabilizing data link protocols, the lack of this knowledge (or an approximation of it) precludes the existence of bounded-size self-stabilizing deterministic data link solutions. In other words, when no bound is known, the use of data link protocols does not provide fully practical solutions for the design of efficient self-stabilizing protocols. Therefore, one has to focus on self-stabilization for message-passing systems, without using data links.

1.2 Message-Passing Model.

Self-stabilizing protocols that operate in message passing systems with unknown initial link capacity and unbounded capacity links are the most versatile, since they can directly be executed in newly set up networks whose characteristics were unknown when the protocol was designed. However, previous works that do not rely on a data link layer require large messages, large memory, or both. For instance, the versatile *census protocol* in [13] collects the entire graph topology at each node, and therefore requires message and memory of $O(n\Delta \log n)$ bits in n-node networks with maximum degree Δ. Similarly, the versatile technique based on so-called *r-operators*, where the algebraic properties of the executed protocol guarantees self-stabilization, has been shown to be quite efficient in general [12, 19, 20]. However, to our knowledge, in the message passing setting, there exists r-operators only for variants of tree construction tasks [12]. Moreover, while the size of the messages used by such protocols remains in $O(\log n)$ bits, the memory at each node might grow as much as $\Omega(\Delta \log n)$ bits.

[1]

1.3 Local tasks, Vertex coloring, and Maximal Independent Set.

One may wonder whether the large amount of memory and communications resources used in the context of unknown and unbounded capacity links is due to the global nature of the task to be solved (census, tree construction). Hence, an intriguing question arises: do local tasks yield high resource consumption when solved self-stabilizingly in an asynchronous setting with unbounded capacity links and unknown initial number of messages? A benchmarking local task in the domain of self-stabilization is that of vertex coloring. In vertex coloring, every process in the network must maintain a color variable such that, for every two adjacent nodes, the value of their color variables is distinct. Typically, the number of colors is supposed to be restricted in the range $\{1, 2, \ldots, \Delta + 1\}$ in networks with nodes with maximum degree Δ. Vertex coloring is one of the most studied tasks in distributed network computing in general, and in self-stabilization in particular, as witnessed by numerous contributions: [5, 7–9, 23, 26, 28, 36–38]. While most previous work about self-stabilizing vertex coloring considered the state model (see [8, 9, 23, 26, 38]), a few paper considered the message passing model [5, 28, 36, 37]. However, all of the existing solutions for this latter model provide probabilistic guarantees only [28, 36, 37], except the recent contribution by Barenboim et al [5]. Moreover, most of them assume some strong or weak forms of a synchronous execution model [5, 28, 37]. Shortly put, to our knowledge, there are no deterministic self-stabilizing vertex coloring protocols that operate in the *asynchronous* message passing setting, where the number of initial spurious messages in communication links is *unknown* to the participating processes, and the link capacity is unbounded.

General $(\Delta + 1)$-node coloring in a distributed fashion was initiated by Linial [34] in the synchronous LOCAL model. Linial showed that in constant-degree graphs the $(\Delta + 1)$-coloring problem requires $\Omega(\log^* n)$ rounds. While Linial does not study the space complexity of this problem, the LOCAL model requires each node to send its neighborhood messages carrying its unique ID, yielding a $\Omega(\log n)$ lower bound on every message size. Focusing on the upper bounds in the same model, the best known result for deterministic algorithms in general graphs are $O(\log^2 \Delta \log n)$ rounds [22] and $\widetilde{O}(\sqrt{\Delta} + \log^* n)$ rounds [4]. The first one [22] requires messages of size $O(\log n)$ bits and $O(\Delta \log n)$ bits per process. The second one [4] requires $O(\log n + \Delta \log \Delta)$ bits per process. Self-stabilizing algorithms considered the asynchronous setting, counting the number of individual node steps to reach a $(\Delta + 1)$-node coloring. The general solution by Gradinariu and Tixeuil [26] requires $O(\Delta n)$ steps and $O(\log \Delta)$ bits by process. In unidirectional general networks, deterministic self-stabilizing solutions [8] are $\Omega(n(n-1)/2)$ steps, while probabilistic ones [7] are $O(\Delta n)$ steps in expectation. Both algorithms require only $O(\log \Delta)$ bits. In the last three papers [7, 8, 26], the state model is used. This implies that processes can read the entire memory of their neighbors freely and atomically, hence there is no copying of relevant information (e.g. their neighbor ID or color) in their own memory.

A *Maximal Independent Set (MIS)* is a set of processes such that no two neighboring processes belong to the set (Independent Set), and the set is maximal (*i.e.*, it is not the subset of another independent set). While the centralized sequential solution to this local task is simple, the first studies considered the

design of fast parallel MIS construction protocols [31, 35]. In the distributed version of the problem, processes maintain a Boolean variable stating whether they belong to the MIS [6, 21]. Most of the self-stabilizing algorithms for this problem consider the state model [3, 27, 30, 40]. To our knowledge, the only solution considering the message-passing model is the one proposed by Goddard *et al.* [24], however this protocol considers a synchronous setting. It stabilizes in $O(n)$ synchronous rounds and $O(n^2)$ (individual node) steps. It requires $O(\Delta \log n)$ bits of memory per process and the exchanged messages have a size of at least $O(\log n)$ bits. The state model protocols [3, 27, 40] operate under an asynchronous scheduler. All protocols require $O(n)$ steps to construct the MIS and a memory of size $O(1)$ bits (yet they require to compare the (whole) id of a process and its neighbors at each step, which is a free operation in the state model).

To design our coloring and MIS algorithms, we use a *Directed Acyclic Graph (DAG)* construction protocol as a building block. Building a DAG in a probabilistic way requires only a constant expected stabilization time, assuming a wireless communication model [28, 37]. However no deterministic bounds have been provided in this model. Ghosh and Karaata [23] proposed a deterministic self-stabilizing DAG construction over planar graphs. However, its complexity was not studied, and remains unknown.

1.4 Our contribution.

Until this paper, it remained unknown whether resource-efficient self-stabilizing solutions exist when communication links have unbounded capacity and an unknown initial number of messages. We show that, for some local tasks, the answer to this question is positive.

In more detail, we establish the following result: without any assumption on the number of messages present initially (and their content), starting from any configuration, we present protocols that are self-stabilizing for the tasks of vertex coloring and MIS construction. They use $O(\log \log n + \Delta \log \Delta)$ bits of memory per node and $O(\log \log n + \log \Delta)$ bits of information per message, where n denotes the number of nodes in the network and Δ its maximum degree. Notice that we consider an upper bound k on the initial number of messages in the communication links. This upper bound is only used to analyze the number of exchanged messages of our algorithms, yet it is not necessary for their correctness and convergence.

A key ingredient of our protocols of independent interest is a symmetry-breaking mechanism that locally orients every link in the network so that the overall orientation is acyclic (hence constructing a directed acyclic graph in the network), simplifying the design of higher layer algorithms such as vertex coloring or maximal independent set.

Our work thus paves the way toward resource-efficient self-stabilizing protocols for the most challenging communication model, enabling solutions to remain valid when new networks are considered.

2 Model

The communication model consists of a point-to-point communication network described by a connected graph $G = (V, E)$ where the nodes V represent the processes and the set E represents bidirectional communication channels. Processes communicate by message passing: a process sends a message to another by depositing the message in the corresponding channel. We denote by $N(v)$ the set of processes that are *neighbors* of process v, *i.e.*, there exists a communication link between them and v. Let us denote by n the number of processes and by Δ the degree of the graph. Note that we denote by $\delta(v)$ the degree of node v.

2.1 Communications.

The communication model is asynchronous message passing with FIFO channels (on each link messages are delivered in the same order as they have been sent). The capacity of each link is unbounded. The *initial* number of messages per link is bounded by an integer k, however the nodes do not know k.[2] $Q(u, v) = (m_q, m_{q-1}, \ldots, m_1)$ is the queue representing the messages in FIFO order in the channel from u to v, where m_1 is the head of the queue. We assume each node v is *fair* with respect to its input channels: if v receives an arbitrarily large number of messages, then a given message m cannot stay in v's input channel forever. We denote by $\mathcal{M}$ the set of all possible messages. 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 two neighboring processes can associate different port numbers for the communication link between them. We denote by $\mathsf{Ports}(v)$ the set of port numbers for the adjacent edges of process v. The port number associated by v to the edge (v, u), if it exists, is denoted $\mathsf{pt}(v, u)$.

2.2 Execution.

Each process v maintains some *variables*. We denote by var_v the value of variable var at process v. The *state* of a process is the vector of the values of its variables. A *configuration* is the vector of the states of every process and the content of the channels between every two neighboring processes. We denote by $var_v(\gamma)$ the value of variable var_v in configuration γ. Similarly, $var_v[u, \gamma]$ and $Q(\gamma, v, u)$ denote the value of the entry u of array var_v in γ and the content of channel $Q(v, u)$ in γ, respectively. The set of all configurations is denoted by Γ.

Let $\mapsto$ be the binary relation between configurations such that $\gamma \mapsto \gamma'$ if the system can reach γ' from γ by executing an *(atomic) step*. During a step, some processes: (a) receive messages (at most one by incoming channel), (b) do some internal computation, and (c) send some messages (at most one by outgoing channel). An *execution* is a maximal sequence of configurations $e = \gamma_0, \gamma_1, \ldots, \gamma_i, \ldots$ such that $\forall i > 0$, $\gamma_{i-1} \mapsto \gamma_i$. Configuration γ_0 is the *initial configuration* of e.

[2] This bound k is only used in the complexity study, in order to count the number of exchanged messages. However, it is not needed for the correctness and convergence of the proposed algorithms.

2.3 Identifier.

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 $\text{Bit}_v(i)$ function. The function Bit_v returns the places where there is a bit with value one in ID_v. Specifically, $\text{Bit}_v(i)$ returns the position (numbered from right to left) of the ith most significant (*i.e.* (from left to right) bit equal to 1. This function can be hard-coded in the immutable code portion of the node. For example, suppose node v has identifier 10 (in decimal notation), or 1010 (in binary notation). Then, one can implement $\text{Bit}_v(i)$ as follows for $v = 1010$:

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

Since we assume that all identifiers are $O(\log n)$ bits long, the Bit_v function only returns values with $O(\log \log n)$ bits. Also, when executing Function 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. Thus, this allows us to determine the i^th bit of the identifier using only $O(\log \log n)$ bits of mutable memory.

Notice that one can choose to hardcode function Bit_v directly into the code of the algorithm instead of storing the identifier in the immutable memory of the node.

2.4 Self-stabilization.

An algorithm $\mathcal{A}$ is *self-stabilizing* for some specification SP if there exists a subset of *legitimate configurations* $\Gamma_\ell \subseteq \Gamma$ such that:

- **Closure:** Γ_ℓ is closed, *i*.e., $\forall \gamma, \gamma' \in \Gamma$ such that $\gamma \mapsto \gamma'$, if $\gamma \in \Gamma_\ell$ then $\gamma' \in \Gamma_\ell$.
- **Convergence:** $\Gamma \rhd \Gamma_\ell$, *i*.e., for any execution $e = \gamma_0, \gamma_1, \ldots, \gamma_i, \ldots$ of $\mathcal{A}$ starting from an arbitrary initial configuration $\gamma_0 \in \Gamma$, $\exists i \geqslant 0$ such that $\gamma_i \in \Gamma_\ell$.
- **Correctness:** For any execution $e = \gamma_0, \gamma_1, \ldots, \gamma_i, \ldots$ of $\mathcal{A}$ starting from a legitimate configuration $\gamma_0 \in \Gamma_\ell$, e satisfies the specification SP.

2.5 Complexities.

When studying the complexity of a self-stabilizing message-passing algorithm, one usually consider three different complexities: the convergence time, the number of messages exchanged before convergence, and the memory requirement.

For the purpose of time complexity analysis, we define a *round* as the smallest execution fragment such that the two following conditions hold: *(i)* all messages that were in transit at the beginning of the round are received and processed[3] by the nodes, and *(ii)* all nodes that have no message in transit in their input

[3]Processed here means that, if the algorithm expects some treatment on the reception of a message, such treatment has been executed.

channels trigger a timeout and process it (*i.e.,* execute the code handling the timeout).

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). A possible explanation is that every node can be modeled as an automaton whose states are the possible states of memory and whose transitions are deduced from the code of the algorithm (that is deterministic in our case). If a deterministic automaton has x states, the number of arcs cannot be greater than x^2. So, if the node uses $O(\log(x))$ bits of memory for the states, it also uses $O(2\log(x)) = O(\log(x))$ bits for the code. Thus, to design a space-efficient self-stabilizing algorithm, one has to make sure that the program counter does not grow beyond limits.

3 Algorithm DAG

The first layer of our solution is to provide a symmetry-breaking mechanism. Our approach is to construct a directed acyclic graph (or DAG) based on the unique identifiers of the nodes: hence an edge is to be oriented from the lower identifier to the higher identifier. Of course, neighboring nodes do *not* know the identifier of the other, and should not communicate the identifier directly to each other as it would break the $o(\log n)$ bits constraint on messages.

3.1 Description of Algorithm DAG

A formal description of our algorithm is presented as Algorithm 1. In a rougher way, for each adjacent link (v, u) of process v, v maintains a binary variable $\mathsf{Ord}_v(p)$, where $p = \mathsf{pt}(v, u)$, as follows. $\mathsf{Ord}_v(p)$ represents the orientation of (v, u). More precisely, $\mathsf{Ord}_v(p)$ equals 0 if v's identifier is greater than the identifier of u and 1 otherwise. To update variable $\mathsf{Ord}_v(p)$, v continuously exchanges information about its identifier with u in a compact way using the Bit_v function and a counter variable $\mathsf{cnt}_v \in \{1, \ldots, \lceil \log ID_v \rceil\}$ of size $O(\log \log n)$ bits. Figure 1 shows the DAG that is built on an example.

When the counter equals to 1, v sends a message to each of its neighbors u requesting for the position of its own first bit of value one (see Line 12). A neighbor u replies by a message $< 1, B >$, where B is the position of the first bit with value one of u (see Line 14). Now, if $B > \mathrm{Bit}_v(1)$, u's identifier is greater than that of v, and if $B < \mathrm{Bit}_v(1)$, then u's identifier is smaller than that of v. Finally, if $B = \mathrm{Bit}_v(1)$, the comparison should continue. When v has received answers from every neighbor concerning the first position, v increments its counter (see Line 8). Then, for each neighbor whose status remains unknown (that is, the condition $B = \mathrm{Bit}_v(1)$ has been satisfied), v requests the value of the second bit, and so on until all link orientations are established. When this is the case and the counter has reached the length of the binary identifier, the

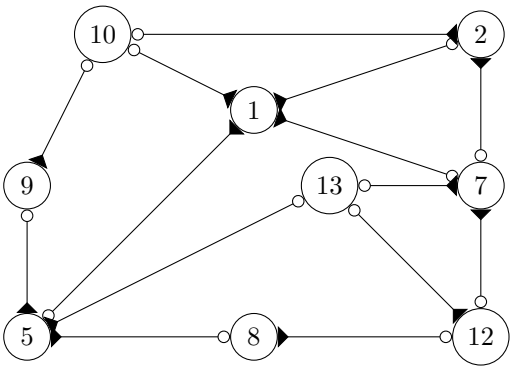

Figure 1: DAG: $\mathsf{Ord}_v(p) = 0$ is represented by a circle and $\mathsf{Ord}_v(p) = 1$ is represented by a black triangle.

comparison process restarts, to achieve that the node executes the $Restart(v)$ function (see Lines 1-4) which, in particular, sets cnt_v to 1.

Due to the arbitrary initial configuration, where the link may contain an unknown number of corrupted messages, this process repeats indefinitely. A **Do forever** process is used to handle the case of a deadlock due to lack of messages (see Lines 25-26). Moreover, messages $< \ell, B >$ which are received by v when its counter cnt_v is different than ℓ are discarded.

In addition to variables $\mathsf{Ord}_v(p)$ and cnt_v, process v maintains the following variables. The variable $\mathsf{tmp}_v(p) \in \{0, 1, \perp\}$ is a temporary variable used during the computation of the order of the identifiers. The assignment of the variable $\mathsf{tmp}_v(p)$ to $\perp$ means that the orientation of the link (v, u) (if $p = \mathsf{pt}(v, u)$) has not yet been computed at this step of the algorithm. It is important to note that, contrary to the $\mathsf{tmp}_v(p)$ variable, the $\mathsf{Ord}_v(p)$ variable does not take the $\perp$ state. Indeed, once the algorithm has converged, in spite of the permanent recalculation, the DAG is maintained thanks to this variable which does not change its value anymore. Moreover, the variable wait_v is a set of port numbers of the node v used to remember the neighbors that have not yet responded to v during the current step of the DAG calculation.

Some privacy In addition to reducing the memory requirements, using compact identifiers for this algorithm can provide some privacy to the nodes. Indeed, since the nodes will never exchange their whole identifier at a time, an adversary temporarily listening to the network and capturing a few messages will only learn a part of their identifiers. Even the nodes will, on average, not learn the whole identifier of their neighbors. Indeed, they only send the bits that are necessary for their comparison. In the worst case, the identifier of two neighboring nodes differ only on their least significant bits, so they must discover the whole identifier of their neighbor to choose the orientation of their common edge. However, in other cases, only some most significant bits are sufficient to decide which neighbor has the greatest identifier.

3.2 Correctness of Algorithm DAG

We now state the main result about Algorithm 1:

Algorithm 1: DAG Algorithm

```
1  Function Restart(v) is
2      cnt_v := 1; wait_v := Ports(v)
3      forall p ∈ Ports(v):
4          tmp_v[p] := ⊥
5  Function Step(v) is
6      if wait_v = ∅ then
7          if cnt_v < ⌈log ID_v⌉ then
8              cnt_v := cnt_v + 1
9              wait_v := {p ∈ Ports(v) : tmp_v[p] = ⊥}
10         else
11             Restart(v)
12     send < cnt_v > to ∀p ∈ wait_v
13 Upon receipt of < ℓ > from port p
14     send < ℓ, Bit_v(ℓ) > to p
15     Step(v)
16 Upon receipt of < ℓ, B > from port p
17     if (p ∈ wait_v) ∧ (ℓ = cnt_v) then
18         wait_v := wait_v \ {p}
19         if B > Bit_v(ℓ) then
20             Ord_v[p] := 1; tmp_v[p] := 1
21         if B < Bit_v(ℓ) ∨ (B = ∅) then
22             Ord_v[p] := 0; tmp_v[p] := 0
23     Step(v)
24 Do forever
25     if ∃p ∈ Ports(v) : (p = pt(v, u)) ∧ (Q(u, v) = ∅) then
26         Step(v)
```

Theorem 1. *Algorithm 1 solves the spanning DAG construction problem in a self-stabilizing manner in n-nodes graphs with maximum degree Δ, assuming the message passing model. If the n node identifiers are in $[1, n^c]$ for some $c \geqslant 1$, then Algorithm 1 uses $O(\log\log n + \Delta \log \Delta)$ bits of memory per node and $O(\log\log n)$ bits per message. Moreover, it converges after $O(\log n)$ rounds, and the exchange of $O(k\Delta n \log n)$ messages, where k (unknown to the algorithm) is the maximum number of (potentially corrupted) messages initially present in each communication link.*

An *erroneous* message is a message that does not match the identifier of the putative sender node. More formally, the message $< \ell, B >$ in queue $Q(v, u)$ is erroneous if and only if $B \neq \text{Bit}_u(\ell)$. To prove Theorem 1, we first prove that erroneous messages eventually disappear, and that the algorithm never creates a erroneous message (Lemma 1).

Let us first define some functions used by Lemma 1. Let $\lambda_m : \Gamma \times \mathcal{M} \times V \times V \to \mathbb{N}$ be the following function:

$$\lambda_m(\gamma, m, u, v) = \begin{cases} 1 & \text{if } m =< \ell, B > \wedge (B \neq \text{Bit}_u(\ell)) \\ 0 & \text{otherwise} \end{cases}$$

This function captures if a message in the link from u to v is erroneous. The λ_Q function captures the number of erroneous messages in the link between u and v. Let $\lambda_Q : \Gamma \times V \times V \to \mathbb{N}$ be the following function:

$$\lambda_Q(\gamma, u, v) = \sum_{m \in Q(\gamma, u, v)} \lambda_m(\gamma, m, u, v)$$

Let $\lambda : \Gamma \times V \to \mathbb{N}$ and $\Lambda : \Gamma \to \mathbb{N}$ be the following functions that respectively capture the number of erroneous messages that v can receive and the number of erroneous messages for the entire configuration:

$$\lambda(\gamma, v) = \sum_{u \in N(v)} \lambda_Q(\gamma, u, v) \text{ and } \Lambda(\gamma) = \sum_{v \in V} \lambda(\gamma, v)$$

Finally, we define the set of configurations $\Gamma_{\mathcal{B}}$ as follows $\Gamma_{\mathcal{B}} = \{\gamma \in \Gamma : \Lambda(\gamma) = 0\}$

Lemma 1. *$\Gamma \rhd \Gamma_{\mathcal{B}}$ in $O(1)$ rounds and $O(k\Delta n)$ messages, and $\Gamma_{\mathcal{B}}$ is closed.*

Proof. In configuration $\gamma_i \in \Gamma$, when node v receives an erroneous message $< \ell, B >$ from u with $B \neq \text{Bit}_u(\ell)$, v sends in configuration γ_j with $j > i$ a message $< cnt_v >$ (see Line 23 and Function *Step*(v) of Algorithm 1). As a consequence, since the erroneous message is no longer in $Q(\gamma_i, u, v)$, we obtain $\lambda_Q(\gamma_i, u, v) < \lambda_Q(\gamma_j, u, v)$. Note that, we obtain $\lambda_Q(\gamma, u, v) = 0$ after at most k receptions of messages through the port p of node v (p being the port number of v leading to u). So, as there at most k initially erroneous messages in each link, a configuration in $\Gamma_{\mathcal{B}}$ is reached in at most one round. Since a node u processes at most $k\delta(u)$ messages per round, the total number of messages sent before a configuration in $\Gamma_{\mathcal{B}}$ is reached in $O(k\Delta n)$.

If node u receives message $< \ell >$ in γ_i, it sends $< \ell, \text{Bit}_u(\ell) >$ in configuration γ_x (see Line 13 of Algorithm 1) followed by a message $< cnt_v >$ in γ_y (see

Function $Step(v)$ of Algorithm 1), with $i < x < y$. So, u never sends $< \ell, B >$ with $B \neq \text{Bit}_u(\ell)$. As a result,

$$\lambda_Q(\gamma_i, u, v) < \lambda_Q(\gamma_j, u, v) = \lambda_Q(\gamma_x, u, v) = \lambda_Q(\gamma_y, u, v) \text{ and } \Lambda(\gamma_i) < \Lambda(\gamma_y).$$

So, $\Gamma_{\mathcal{B}}$ is closed.

□

In order to define a configuration that satisfies a DAG property, we need to define some functions and predicates. The function $dif(v, u)$ returns the minimum bit position that differentiates the identifier of v from that of u. More formally:

$$dif(v, u) = \min\left\{\ell : \ell \in \{1, \ldots, \lceil \log n \rceil\} \wedge \text{Bit}_v(\ell) \neq \text{Bit}_u(\ell)\right\} \tag{1}$$

Now, the following predicates capture whether a node v and its neighbor u satisfy our DAG condition in configuration γ. The predicate G_l captures the behavior of variable tmp when the counter cnt_v has not yet reached the bit that allows differentiating the identifiers (*i.e.*, when cnt_v is *lower* than or equal to $dif(u, v)$).

$$G_l(\gamma, u, v) \equiv (\mathsf{cnt}_v(\gamma) \leqslant dif(u, v)) \wedge (\mathsf{tmp}_v[\gamma, u] = \bot) \tag{2}$$

The predicate G_g captures the behavior of the variable tmp when the counter cnt_v has reached the bit that allows differentiating the identifiers. In other words, when the node v knows it is greater or smaller than its neighbor u. Note that, at this step, the u node has responded so the port must not be in the wait_v set, which represents the set of ports that have not responded.

$$G_g(\gamma, v, u) \equiv \mathsf{cnt}_v(\gamma) \geqslant dif(u, v)) \wedge (\mathsf{tmp}_v[\gamma, u] \in \{0, 1\}) \wedge \mathsf{pt}(v, u) \notin \mathsf{wait}_v(\gamma) \tag{3}$$

The G predicate captures the behavior of variable tmp in both cases.

$$G(\gamma, u, v) \equiv G_l(\gamma, u, v) \vee G_g(\gamma, u, v) \tag{4}$$

Let $\phi_p(\gamma, u, v) : \Gamma \times V \times V \to \mathbb{N}$ be the following function:

$$\phi_p(\gamma, u, v) = \begin{cases} 0 & \text{if } (\text{ID}_u > \text{ID}_v) \wedge (\mathsf{Ord}_v[\gamma, u] = 1) \wedge G(\gamma, v, u) \\ 0 & \text{if } (\text{ID}_u < \text{ID}_v) \wedge (\mathsf{Ord}_v[\gamma, u] = 0) \wedge G(\gamma, v, u) \\ 1 & \text{otherwise} \end{cases} \tag{5}$$

Let $\phi : \Gamma \times V \to \mathbb{N}$ and $\Phi : \Gamma \to \mathbb{N}$ be the following potential functions:

$$\phi(\gamma, v) = \sum_{u \in N(v)} \phi_p(\gamma, u, v) \qquad \text{and} \qquad \Phi(\gamma) = \sum_{v \in V} \phi(\gamma, v) \tag{6}$$

Finally, we define the set of configurations $\Gamma_{\mathtt{DAG}}$ as the following $\Gamma_{\mathtt{DAG}} = \{\gamma \in \Gamma_{\mathcal{B}} : \Phi(\gamma) = 0\}$

Lemma 2. $\Gamma_{\mathcal{B}} \rhd \Gamma_{\mathtt{DAG}}$.

Proof. Let us consider a configuration $\gamma_0 \in \Gamma_{\mathcal{B}}$ and a node v such that $\phi(\gamma_0, v) > 0$, which has as a direct consequence that there exists a node $u \in N(v)$ such that $\phi_p(\gamma_0, u, v) = 1$.

To reach $\phi_p(\gamma, u, v) = 0$, the node v must have a correct value in its variable $\mathsf{Ord}_v[u]$. To achieve this, it must compare its identifier with that of u. The only way to compare all the bits of the identifiers of u and v is to increment the counter of v. The counter of v is incremented in the *Step* function if and only if $\mathsf{wait}_v = \varnothing$. In the following, we prove that starting from configuration $\gamma_0 \in \Gamma_{\mathcal{B}}$ with $\mathsf{wait}_v \neq \varnothing$, the system converges to a configuration $\gamma \in \Gamma_{\mathcal{B}}$ with $\mathsf{wait}_v = \varnothing$.

Let us assume that, $\mathsf{wait}_v \neq \varnothing$ in the configuration γ_0 . It means that there exists a neighbor u of v such that $p \in \mathsf{wait}_v$ and $p = \mathsf{pt}(v, u)$. Port p is removed from the set wait_v when v receives message $m_u^{\ell_v, B} =< \ell, B >$ from u (see Line 18 of Algorithm 1). Moreover, $\gamma \in \Gamma_{\mathcal{B}}$ guarantees that, when v receives a message of type $< \ell, B >$ from u then $B = \mathrm{Bit}_u(\mathsf{cnt}_v(\gamma))$. Several cases must be considered to capture the removal of p from wait_v (see Figure 2).

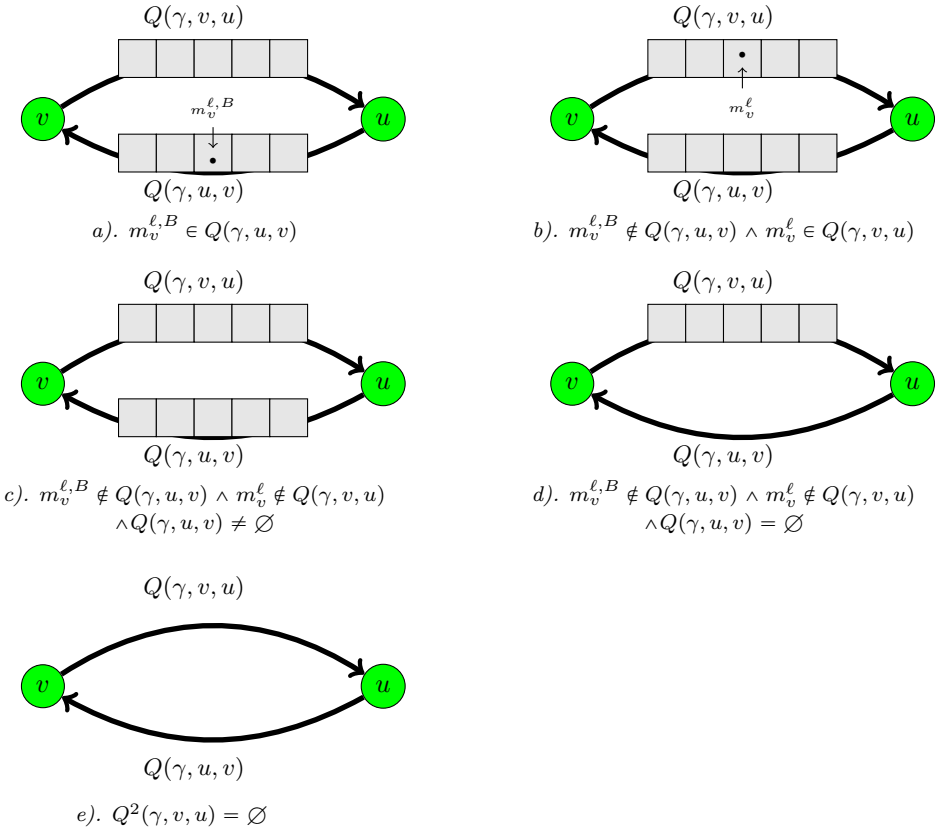

Figure 2: The different cases before the removal of p from wait_v

1. First, assume that $m_u^{\ell_v, B}$ is in $Q(\gamma_0, u, v)$. Let us define the function $I_A(\gamma, u, v)$ that captures the number of message receipts required to achieve

the removal of p from wait_p, $I_A : \Gamma \times V \times V \to \mathbb{N}$:

$$I_A(\gamma, u, v) = \min\{i : m_i \in Q(\gamma, u, v) \wedge m_u^{\ell_v, B} = m_i\} \tag{7}$$

Notice that we assume a numbering of the messages inside the channels in a FIFO way, just for the reasoning. More formally, if $m_v^{\ell,B} \in Q(\gamma_j, u, v)$ we obtain $I_A(\gamma_j, u, v) > I_A(\gamma_{j'}, u, v)$ where $j' > j$ and v received a message from u in $\gamma_{j'}$.

2. If $m_u^{\ell_v,B} \notin Q(\gamma_j, u, v)$, u must receive a message $m_v^{\ell_v} =< \mathsf{cnt}_v(\gamma) >$ from v in order to send a message $m_v^{\ell,B}$ in configuration $\gamma_{j'}$ with $j' > j$ (see Line 12 of Algorithm 1). So let us suppose that, $m_u^{\ell_v,B} \notin Q(\gamma_0, u, v)$ and $m_v^{\ell_v} \in Q(\gamma_0, v, u)$. The function $b : \Gamma \times V \times V \to \mathbb{N}$ captures the number of messages that a node can receive before it sends a message of type $m_u^{\ell_v,B}$ in this case.

$$b(\gamma, v, u) = \min\{i : m_i \in Q(\gamma, v, u) \wedge m_u^{\ell_v} = m_i\} \tag{8}$$

Remark that, when a node u receives a message, u replies by sending either one message (if the received one is of type $< \ell, B >$, see Lines 23 and 12 of Algorithm 1) or two messages (if the received one is of type $< \ell >$, see Lines 12, 14, and 15 of Algorithm 1). Moreover, a configuration in $\Gamma_{\mathcal{B}}$ does not guarantee that the messages of type $< \ell >$ are equal to m_v^{ℓ}. Let us define the function $I_B(\gamma, u, v) : \Gamma \times V \times V \to \mathbb{N}$ that captures the number of messages v needs to receive to achieve the removal of p from wait_p:

$$I_B(\gamma, u, v) = 2b(\gamma, v, u) + |Q(\gamma, u, v)| \tag{9}$$

3. Suppose now, $m_u^{\ell_v,B} \notin Q(\gamma_0, u, v)$ and $m_v^{\ell_v} \notin Q(\gamma_0, v, u)$ but $Q(\gamma_0, u, v) \neq \varnothing$. Remark that, upon each reception of a message, v executes *Step(v)* (see Lines 15 and 23 in Algorithm 1) which in turn sends a message m_v^{ℓ}. So after $I_C(\gamma, u, v)$ messages receptions by v, v executes $\mathsf{wait}_v := \mathsf{wait}_v \backslash \{p\}$, where

$$I_C(\gamma, u, v) = 1 + |2Q(\gamma, v, u)| + |Q(\gamma, u, v)| \tag{10}$$

4. Suppose now, $m_u^{\ell_v,B} \notin Q(\gamma, u, v)$, $m_u^{\ell_v} \notin Q(\gamma, v, u)$ and $Q(\gamma, u, v) = \varnothing$ but $Q(\gamma, v, u) \neq \varnothing$. In this case, the node v execute **Do forever** and sends a message m_v^{ℓ} (see Lines 24, 25, 26, and 12 in Algorithm 1). So after $I_C(\gamma, u, v)$ messages received by v, v executes $\mathsf{wait}_v := \mathsf{wait}_v \backslash \{p\}$.

5. If $Q^2(\gamma, u, v) = \varnothing$, where $Q^2(\gamma, u, v)$ denote the set of messages contained in $Q(\gamma, u, v)$ and $Q(\gamma, v, u)$, both u and v can execute **Do forever**. The maximum number of receptions, *i.e.*, 2, is obtained when u and v jointly execute **Do forever**.

6. To summarize the following function captures all aforementioned cases, let $d(\gamma, u, v) : \Gamma \times V \times V \to \mathbb{N}$ be :

$$d(\gamma, u, v) = \begin{cases} I_A(\gamma, u, v) & \text{if } m_u^{\ell_v,B} \in Q(\gamma, u, v) \\ I_B(\gamma, u, v) & \text{if } m_u^{\ell_v,B} \notin Q(\gamma, u, v) \wedge m_v^{\ell_v} \in Q(\gamma, v, u) \\ I_C(\gamma, v, v) & \text{if } m_u^{\ell_v,B} \notin Q(\gamma, u, v) \wedge m_v^{\ell_v} \notin Q(\gamma, v, u) \\ & \qquad \wedge Q^2(\gamma, u, v) \neq \varnothing \\ 2 & \text{if } Q^2(\gamma, u, v) = \varnothing \end{cases} \tag{11}$$

and let $D(\gamma, v) : \Gamma \times V \to \mathbb{N}$ be the following potential function:

$$D(\gamma, v) = \sum_{\mathsf{pt}(v,u) \in \mathsf{wait}_v(\gamma)} d(\gamma, u, v) \tag{12}$$

By construction of $d(\gamma, u, v)$, upon each reception of a message by v in configuration γ', we obtain $d(\gamma, u, v) > d(\gamma', u, v)$. So the system reaches a configuration $\gamma_e \in \Gamma_{\mathcal{B}}$ such that $D(\gamma_e, v) = 0$, in other words $\mathsf{wait}_v(\gamma_e) = \varnothing$.

Now we must capture the number of messages received by v to increase its counter until it puts the right values in $\mathsf{Ord}_v[u]$, namely v reaches a configuration $\gamma \in \Gamma_{\mathcal{B}}$ where $\phi_p(\gamma_0, u, v) = 1$. A node v executes the algorithm if it receives a $< \ell, B >$ message, a $< \ell >$ message or in the absence of a message v executes the function **Do forever**. In all cases, in γ_a (with $a > 0$), the node v executes the *Step*(v) function (see Lines 15, 23, and 26 in the Algorithm 1). In the γ_e configuration where $\mathsf{wait}_v(\gamma_e) = \varnothing$, the v node increases its own counter or executes the *Restart(v)* function (see Lines 8, 11 in the Algorithm 1).

Note that, when v executes *Restart(v)* in configuration γ_g the function $G_l(\gamma_g, u, v)$ becomes true, because the counter of v is set to 1 and the variable $\mathsf{tmp}_v[p]$ is set to $\perp$ for all ports of v (see Lines from 2 to 4 in the Algorithm 1).

Moreover, starting from a configuration $\gamma_r \in \Gamma_{\mathcal{B}}$ where the node v executes *Restart(v)*, every message $m =< \ell, B >$ in $Q(\gamma_{r'}, u, v)$, with $r' > r$, satisfies $\mathrm{Bit}_u(\ell) = B$. Then, when cnt_v reaches $dif(u, v)$ in γ_j, with $j > r$, the variables $\mathsf{Ord}_v[\gamma_j, u]$ and $\mathsf{tmp}_v[\gamma_j, u]$ takes values 1 if $\mathrm{ID}_u > \mathrm{ID}_v$, or 0 if $\mathrm{ID}_u < \mathrm{ID}_v$ (see Lines 17 and 18 of Algorithm 1). In addition, $\mathsf{pt}(v, u) \notin \mathsf{wait}_v(\gamma_j)$ so $G_g(\gamma_j, v, u)$ is true and $\phi_p(\gamma, u, v)$ reaches 0.

To summarize, starting from a configuration γ_0 where $\phi_p(\gamma_0, u, v) = 1$, $\phi_p(\gamma, u, v)$ reaches 0 after at most $\lceil \log \mathrm{ID}_v \rceil - \mathsf{cnt}_v(\gamma) + dif(u, v)$ increments of cnt_v. Note that, if in the initial configuration γ_0, $G(\gamma_0, u, v)$ is true, the counter only needs $dif(u, v) - cnt_v(\gamma)$ increments. To capture that we define function $\epsilon : \Gamma \times V \times V \to \mathbb{N}$:

$$\epsilon(\gamma, u, v) = \begin{cases} \lceil \log \mathrm{ID}_v \rceil - \mathsf{cnt}_v(\gamma) + dif(u, v) & \text{if } \mathsf{cnt}_v > dif(u, v) \\ dif(u, v) - \mathsf{cnt}_v & \text{otherwise} \end{cases} \tag{13}$$

The following function capture all the process of the Algorithm 1, $\alpha : \Gamma \times V \to \mathbb{N} \times \mathbb{N}$:

$$\alpha(\gamma, v) = (\epsilon(\gamma, v), D(\gamma, v))$$

and $\alpha(\gamma', v) < \alpha(\gamma, v)$ if and only if $\epsilon(\gamma', u, v) < \epsilon(\gamma, u, v)$ or $\epsilon(\gamma', u, v) = \epsilon(\gamma, u, v)$ and $d(\gamma', u, v) < d(\gamma', u, v)$. Let $\gamma_i \in \Gamma_{\mathcal{B}}$. In a configuration γ_{i+1} where v receives at least one message, we have $\alpha(\gamma_{i+1}, u, v) < \alpha(\gamma_i, u, v)$. When $\alpha(\gamma_j, u, v) = (0, 0)$ the function $\phi(\gamma_j, u, v)$ decreases by one. Note that after

that the function α can increase because the computation of the algorithm is perpetual but the function ϕ does not increase because $\mathsf{Ord}_v[u]$ does not change anymore.

So, the system reaches a configuration γ where $\Phi(\gamma) = 0$, in other words $\gamma \in \Gamma_{DAG}$ and $\Gamma_{\mathtt{DAG}}$ is closed. □

Lemma 3. *Algorithm 1 converges in $O(\log n)$ rounds, and exchanges before convergence $O(k\Delta n \log n)$ messages .*

Proof. To converge from Γ to $\Gamma_{\mathcal{B}}$, it requires receiving and disposing of all erroneous messages in the channels as shown in the proof of Lemma 1. Each process must treat at most k messages and in one round, the system reaches a configuration of $\Gamma_{\mathcal{B}}$.

Then, we use the potential function Φ to compute the number of rounds. Remember that $\alpha(\gamma, v) = (\epsilon(\gamma, v), D(\gamma, v))$. From a configuration γ where $D(\gamma, v) \neq 0$, process v has to send $O(\Delta k)$ messages (at most $3k + 1$ for each channel) to reach a configuration γ' where $D(\gamma, v) \neq 0$. It lasts at most 4 rounds (case *c).* in the proof of Lemma 2 is the longer one). Function $\epsilon(\gamma, v)$ is bounded by $\log n$. Thus, the system converges from $\Gamma_{\mathcal{B}}$ to Γ_{DAG} in at most $4 \log n$ rounds.

Hence, in at most $4 \log n + 1$ round, each node v computes its set of link directions in the DAG. At each round, v reads $k\delta(v)$ messages, so the system converges after $O(k\Delta n \log n)$ messages. □

Proof of Theorem 1 We first proved that erroneous messages eventually disappear, and the Algorithm 1 never creates an erroneous message (Lemma 1). Starting from a configuration γ in $\Gamma_{\mathcal{B}}$ the set of configurations where there are no erroneous message Algorithm 1 converges to a configuration $\gamma' \in \Gamma_{\mathtt{DAG}}$. $\Gamma_{\mathtt{DAG}}$ is the set of configurations where the variables Ord_v for all $v \in V$ respect $\mathsf{Ord}_v[u] = 1$ if $(\text{ID}_u > \text{ID}_v)$, or $\mathsf{Ord}_v[u] = 0$ if $(\text{ID}_u < \text{ID}_v)$. The convergence and closure in configurations $\Gamma_{\mathtt{DAG}}$ is given by Lemma 2. Finally, by Lemma 3, we have proof that Algorithm 1 converges in $O(\log n)$ rounds, and exchanges $O(k\Delta n \log n)$ messages before convergence.

4 Vertex Coloring

We now present our self-stabilizing $(\Delta+1)$ vertex coloring algorithm, its pseudo-code is given in Algorithm 2. This algorithm is built on top of Algorithm 1, namely we assume that each node v knows which neighbor has a lower identifier, and which has a higher one using variables Ord of the Algorithm 1.

4.1 Description of the Vertex Coloring Algorithm

Each node v maintains a color variable, denoted by $\mathsf{c}_v \in \{1, \ldots, \delta(v) + 1\}$. In addition, v maintains an array with all the known colors of its neighbors denoted $\mathsf{c}_v[u] \in \{1, \ldots, \Delta + 1\}$ for each port u. To do so, infinitely often, v sends its own color to its neighbors by sending a message $< \mathsf{c}_v >$ (see Lines 6, 8, 14, and 20 of Algorithm 2). If v has a neighbor u whose color is identical (in other words, if v detects a conflict) but v's identifier is greater, v changes its color to the

minimum color not used by its neighbors, by executing function $Conflict(v)$ (see Lines 1-8 of Algorithm 2).

Note that, since the maximum degree of the graph is Δ, the size of $\mathsf{c}_v[]$ is bounded by $\Delta \log \Delta$.

Algorithm 2: Vertex coloring

```
1  Function Conflict(v) is                                   /* c_v[u] = c_v */
2      if (∀w ∈ Ports(v) : (c_v[w] ≠ ⊥) ∧
3      (c_v[w] = c_v) ⇒ (Ord_v[w] = 0)) then
4          c_v := min{1, ..., δ(v)}\{c_v[w] : w ∈ Ports(v)}
5          forall w ∈ Ports(v)
6              send < c_v > to w
7      else
8          send < c_v > to u
9  Upon receipt of < c > from port u
10     c_v[u] := c
11     if c_v[u] = c_v then
12         Conflict(v)
13     else
14         send < c_v > to u
15 Do forever
16     forall u ∈ Ports(v) : Q(u, v) = ∅
17         if c_v[u] = c_v then
18             Conflict(v)
19         else
20             send < c_v > to u
```

4.2 Correctness of the Vertex Coloring Algorithm

Theorem 2. *Algorithm 2 solves the vertex coloring problem in a self-stabilizing manner in n-nodes graph with maximum degree Δ, assuming the message passing model and a spanning DAG. It uses $O(\Delta \log \Delta)$ bits of memory per node, $O(\log \Delta)$ bits per message. Moreover, it converges after $O(n)$ rounds and exchanges $O(\Delta\ k\ n)$ messages.*

In the following proof, we assume that the oriented graph described by variables Ord is a spanning DAG. DAG construction is an independent block used by the input vertex coloring algorithm. Consequently, the complexities computed in this section depend solely on the vertex coloring algorithm.

Lemma 4. *Every time process v executes a step, it sends $< \mathsf{c}_v >$ to every neighbor $u \in N(v)$.*

Proof. Each time v executes a step, for every neighbor $u \in N(v)$, two cases can occur:

- v receives a message from u, so v executes $Conflict(v)$, or sends back $< \mathsf{c}_v >$ to u.
- v executes **Do forever**, so it calls $Conflict(v)$, or sends back $< \mathsf{c}_v >$ to u.

If v calls $Conflict(v)$, either v sends back $< \mathsf{c}_v >$ to u, or v changes its color and sends the new one to every neighbor. □

Let $\alpha_1 : \Gamma \times V \to \mathbb{N}$ and $\alpha_2 : \Gamma \times V \to \mathbb{N}$ be the following functions:

$$\alpha_1(\gamma, v) = |\{u \in N(v) : \mathsf{c}_u = \mathsf{c}_v\}| \qquad \text{and } \alpha_2(\gamma, v) = |\{u \in N(v) : \mathsf{c}_v[u] \neq \mathsf{c}_u\}|$$

Let $\mu : \Gamma \times V \times \{1, \ldots, \Delta\} \to \mathbb{N}$ be the following function:

$$\mu(\gamma, v, c) = \begin{cases} 1 & \text{if } \mathsf{c}_v \neq c \\ 0 & \text{otherwise} \end{cases}$$

Let $\alpha_3 : \Gamma \times V \to \mathbb{N}$ be the following function:

$$\alpha_3(\gamma, v) = \sum_{u \in N(v)} \left(\sum_{<c> \in Q(u,v)} \mu(\gamma, u, c) \right)$$

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

$$A(\gamma) = \sum_{v \in V} \big(\alpha_1(\gamma, v) + \alpha_2(\gamma, v) + \alpha_3(\gamma, v)\big)$$

We define $\Gamma_\alpha = \{\gamma \in \Gamma : A(\gamma) = 0\}$.

Lemma 5. $\Gamma \rhd \Gamma_\alpha$.

Proof. Let $d_s(v)$ be the minimum distance from process v to a source of the DAG (*i.e.*, a process with only outgoing edges). First, we show that in finite time, every process v stops changing its color by induction on $d_s(v)$.

- **Base case:** If $d_s(v) = 0$, then v is a source of the DAG and $\forall u \in N(v)$, $\mathsf{Ord}_v[u] = 1$. Thus, v never changes its color c_v (see Line 3).
- **Induction step:** If $d_s(v) = x + 1$, then $\forall u \in N(v)$, either $\mathsf{Ord}_v[u] = 1$, or $d_s(u) < d_s(v) = x + 1$. By induction hypothesis, in finite time, $\forall u \in N(v)$ such that $d_s(u) \leqslant x$, the value of c_u eventually stops changing. Then, by Lemma 4, u sends $< \mathsf{c}_u >$ to v infinitely often, so eventually $\mathsf{c}_v[u] = \mathsf{c}_u$. Now, either v never changes its color, or it gets a new color different from c_u. In the second case, v is no longer in conflict with any neighbor $u \in N(v)$ such that $d_s(u) < d_s(v)$. Node v can only be in conflict with neighbors $w \in N(v)$ such that $\mathsf{Ord}_v[w] = 1$, and thus v does not change its color anymore (see Line 3). Hence, v can change its color at most once.

Once every process stops changing its color forever, a process v cannot send a color different from its own. Thus, there exists a configuration γ_a such that, in every subsequent configuration γ, for every process v, v does not change its color and $\alpha_3(\gamma, v) = 0$. Moreover, in finite time, processes update their local knowledge about their neighbors' colors (see Lemma 4 and Line 10). Thus,

there is a configuration γ_b after γ_a such that, in every configuration γ after γ_b, for every process v, $\alpha_2(\gamma, v) = 0$.

Finally, we show that in every configuration after γ_b, there is no color conflict. Assume by contradiction that in some configuration γ after γ_b, there is a color conflict. Let us consider the process v in conflict that has the greatest ID among all such processes. For every $u \in N(v)$ such that $\mathsf{c}_v = \mathsf{c}_u$, we have $\mathsf{c}_v[u] = \mathsf{c}_u = \mathsf{c}_v$ and $\mathsf{c}_u[v] = \mathsf{c}_v = \mathsf{c}_u$. Moreover, by assumption, $\mathsf{Ord}_v[u] = 0$ and $\mathsf{Ord}_u[v] = 1$. Thus, in finite time, v calls the function $Conflict$, and changes its color (see Line 3), a contradiction. So, in every configuration γ after γ_b, and for every process $v \in V$, $\alpha_1(\gamma, v) = 0$. Hence, $A(\gamma) = 0$, and $\Gamma \rhd \Gamma_\alpha$. □

Lemma 6. *Γ_α is closed.*

Proof. Let $\gamma \mapsto \gamma'$ such that $\gamma \in \Gamma_\alpha$. Let $v \in V$ and $u \in N(v)$.

First, in γ, $\mathsf{c}_v[u] = \mathsf{c}_u \neq \mathsf{c}_v$. Moreover, every message in $Q(u, v)$ contains color $\mathsf{c}_u \neq \mathsf{c}_v$. Thus, v does not change its color during step $\gamma \mapsto \gamma'$. Hence, $\alpha_1(\gamma', v) = 0$.

Since u and v do not change their color between γ and γ', and since every message in $Q(u, v)$ in γ equals c_u, then the value of $\mathsf{c}_v[u]$ does not change, and remains equal to c_u in γ'. Hence, $\alpha_2(\gamma', v) = 0$.

Finally, u does not change its color between γ and γ', and every message sent by u during step $\gamma \mapsto \gamma'$ equals $< \mathsf{c}_u >$ similarly to every other message in $Q(u, v)$ in γ that u did not process during step $\gamma \mapsto \gamma'$. Thus, $\alpha_3(\gamma', v) = 0$. □

Lemma 7. *Algorithm 2 converges in $O(n)$ rounds and exchanges $O(\Delta k n^2)$ messages before convergence.*

Proof. After one round, the system is purged of all possibly erroneous messages (that is, messages that contain a color that does not match the color of the sender node) contained in the initial configuration. Moreover, by Lemma 4, at each round a process v sends its color $< \mathsf{c}_v >$ to all its neighbors, and those neighbors receive this message during the next round.

Now, consider a process v that changes its color during some round. The last time v changes its color during the round, it knows the real colors of its neighbors, *i.e.*, $\forall u \in N(v)$, $\mathsf{c}_v[u] = \mathsf{c}_u$. So by changing its color, v solves its color conflicts with its neighbors having a lower identity and does not create any new conflict with them. Since a process can change its color only if it is in conflict with a node of lower identity, v will not change its color anymore after this round. Hence, in the worst case, processes stop changing their colors after $O(n)$ rounds and there is no color conflict anymore.

In the worst case, at each round, a process replies to every message sent by its neighbors during the previous round. Thus, $O(\Delta k n^2)$ messages are exchanged. □

To avoid starvation of one of the algorithms, we compose them using a *fair composition* [29], *i.e.*, each process alternatively executes a step of Algorithm 1 and a step of Algorithm 2. As demonstrated in previous work [17], fair composition preserves the property of self-stabilization.

5 Maximal Independent Set

As another direct application of Algorithm 1, we present in this section a maximal independent set algorithm. Its pseudo-code is given in Algorithm 3.

5.1 Description of the Maximal Independent Set Algorithm

Similarly to self-stabilizing maximal independent set algorithms written for the state model [27], the priority to join the maximal independent set is deduced from the node identifiers. In our scheme, we assume that each node v knows which neighbor has a lower identifier, and which has a higher one using variables Ord of the Algorithm 1. Each node v maintains a membership variable, denoted by $\mathsf{m}_v \in \{False, True\}$. In addition, v maintains an array $\mathsf{m}_v[]$ with all the known membership status of its neighbors denoted $\mathsf{m}_v[u] \in \{False, True\}$, for each port u. To do so, infinitely often, v sends its own membership status to its neighbors by sending a message $< \mathsf{m}_v >$. If v has a lower identifier neighbor u whose membership status is *True*, v's membership becomes *False*. If none of v's lower identifier neighbors has membership equal to *True*, v's membership becomes True.

Note that, since the maximum degree of the graph is Δ, the size of $\mathsf{m}_v[]$ is upper bounded by Δ.

Algorithm 3: Maximal Independent Set

```
1  Function Update(v) is
2      if (∀w ∈ Ports(v) : (Ord_v[w] = 0) ⇒ m_v[w] = False)) then
3          m_v := True
4      else
5          if (∃w ∈ Ports(v) : (Ord_v[w] = 0) ∧ m_v[w] = True)) then
6              m_v := False
7      forall w ∈ Ports(v)
8          send < m_v > to w
9  Upon receipt of < m > from port u
10     m_v[u] := m
11     Update(v)
12 Do forever
13     Update(v)
```

5.2 Correctness of the Maximal Independent Set Algorithm

Theorem 3. *Algorithm 3 solves the maximal independent set problem in a self-stabilizing manner in n-nodes graph with maximum degree Δ, assuming the message passing model and a spanning DAG. It uses $O(\Delta)$ bits of memory per*

node, $O(1)$ bits per message. Moreover, it converges after $O(n)$ rounds and exchanges $O(\Delta\ k\ n^2)$ messages.

In the following proof, we assume that the oriented graph described by variables Ord is a spanning DAG. DAG construction is an independent block used by the input vertex coloring algorithm. Consequently, the complexities computed in this section depend solely on the maximal independent set algorithm.

Lemma 8. *Every time process v executes a step, it sends $< \mathsf{c}_v >$ to every neighbor $u \in N(v)$.*

Proof. Each time v executes a step, for every neighbor $u \in N(v)$, two cases can occur:

- v receives a message from u, so v executes $Update(v)$ and sends a message to every neighbor.
- v executes **Do forever**, so v executes $Update(v)$, and sends a message to every neighbor.

□

Let $\beta_0 : \Gamma \times V \to \mathbb{N}$, $\beta_1 : \Gamma \times V \to \mathbb{N}$ and $\beta_2 : \Gamma \times V \to \mathbb{N}$ be the following functions:

$$\beta_0(\gamma, v) = \begin{cases} 1 & \text{if } \mathsf{m}_v = \mathit{False} \wedge \forall \{u \in N(v) : u < v\} \mathsf{m}_v = \mathit{False} \\ 0 & \text{otherwise} \end{cases}$$

$$\beta_1(\gamma, v) = |\{u \in N(v) : \mathsf{m}_u = \mathsf{m}_v = \mathit{True}\}|$$

$$\beta_2(\gamma, v) = |\{u \in N(v) : \mathsf{m}_v[u] \neq \mathsf{m}_u\}|$$

Let $\mu : \Gamma \times V \times \{\mathit{True}, \mathit{False}\} \to \mathbb{N}$ be the following function:

$$\mu(\gamma, v, m) = \begin{cases} 1 & \text{if } \mathsf{m}_v \neq m \\ 0 & \text{otherwise} \end{cases}$$

Let $\beta_3 : \Gamma \times V \to \mathbb{N}$ be the following function:

$$\beta_3(\gamma, v) = \sum_{u \in N(v)} \left(\sum_{<m> \in Q(u,v)} \mu(\gamma, u, m) \right)$$

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

$$A(\gamma) = \sum_{v \in V} \big(\beta_0(\gamma, v) + \beta_1(\gamma, v) + \beta_2(\gamma, v) + \beta_3(\gamma, v)\big)$$

We define $\Gamma_\beta = \{\gamma \in \Gamma : A(\gamma) = 0\}$.

Lemma 9. $\Gamma \rhd \Gamma_\beta$.

Proof. Let $d_s(v)$ be the minimum distance from process v to a source of the DAG (*i.e.*, a process with only outgoing edges). First, we show that in finite time, every process v stops changing its maximal independent set membership by induction on $d_s(v)$.

- **Base case:** If $d_s(v) = 0$, then v is a source of the DAG and $\forall u \in N(v)$, $\mathsf{Ord}_v[u] = 1$. Thus, v changes its membership status at most once (see Line 3).
- **Induction step:** If $d_s(v) = x + 1$, then $\forall u \in N(v)$, either $\mathsf{Ord}_v[u] = 1$, or $d_s(u) < d_s(v) = x + 1$. By induction hypothesis, in finite time, $\forall u \in N(v)$ such that $d_s(u) \leqslant x$, the value of m_u eventually stops changing. Then, by Lemma 8, u sends $< \mathsf{m}_u >$ to v infinitely often, so eventually $\mathsf{m}_v[u] = \mathsf{m}_u$. From this point, v changes its maximal independent set membership status at most once, and if so is no longer in conflict with any of its predecessors in the DAG.

Once every process stops changing its maximal independent set membership status, it can only send its actual membership status. Thus, there exists a configuration γ_a such that, in every subsequent configuration γ, for every process v, v does not change its membership status, and $\beta_3(\gamma, v) = 0$. Moreover, in finite time, processes update their local knowledge about their neighbors' membership status (see Lemma 8 and Line 10). Thus, there is a configuration γ_b after γ_a such that, in every configuration γ after γ_b, for every process v, $\beta_2(\gamma, v) = 0$.

Then, we show that in every configuration after γ_b, there is no membership status conflict. Assume by contradiction that in some configuration γ_c after γ_b, there is a membership status conflict. Let us consider the process v in conflict that has the greatest ID among all such processes. For every $u \in N(v)$ such that $\mathsf{m}_v = \mathsf{m}_u = \mathit{True}$, we have $\mathsf{m}_v[u] = \mathsf{m}_u = \mathsf{m}_v$, and $\mathsf{m}_u[v] = \mathsf{m}_v = \mathsf{m}_u$. Moreover, by assumption, $\mathsf{Ord}_v[u] = 0$ and $\mathsf{Ord}_u[v] = 1$. Thus, in finite time, v calls the function *Update*, and changes its membership status (see Line 6), a contradiction. So, in every configuration γ_c after γ_b, and for every process $v \in V$, $\beta_1(\gamma_c, v) = 0$.

Finally, we show that in every configuration after γ_c, the membership status *True* induces a maximal independent set of the graph. Assume for the purpose of contradiction that in some configuration γ after γ_c, there is a node that could join the maximal independent set but doesn't. Let us consider the process v that has the lowest ID among all such processes. Thus, in finite time, v calls the function *Update*, and changes its membership status (see Line 3), a contradiction. So, in every configuration γ after γ_c, and for every process $v \in V$, $\beta_0(\gamma_c, v) = 0$.

Hence, $A(\gamma) = 0$, and $\Gamma \rhd \Gamma_\alpha$. □

Lemma 10. *Γ_α is closed.*

Proof. Let $\gamma \mapsto \gamma'$ such that $\gamma \in \Gamma_\alpha$. Let $v \in V$ and $u \in N(v)$.

First, in γ, $\mathsf{m}_v[u] = \mathsf{m}_u$. Moreover, every message in $Q(u, v)$ contains membership status m_u. Thus, v does not change its membership status during step $\gamma \mapsto \gamma'$. Hence, $\beta_0(\gamma', v) + \beta_1(\gamma', v) = 0$.

Since u and v do not change their membership status between γ and γ', and since every message in $Q(u, v)$ in γ equals m_u, then the value of $\mathsf{m}_v[u]$ does not change, and remains equal to m_u in γ'. Hence, $\beta_2(\gamma', v) = 0$.

Finally, u does not change its membership between γ and γ', and every message sent by u during step $\gamma \mapsto \gamma'$ equals $< \mathsf{m}_u >$ similarly to every other message in $Q(u, v)$ in γ that u did not process during step $\gamma \mapsto \gamma'$. Thus, $\beta_3(\gamma', v) = 0$.

Hence, the result. □

Lemma 11. *Algorithm 3 converges in $O(n)$ rounds and exchanges $O(\Delta kn^2)$ messages before convergence.*

Proof. After one round, the system is purged of all possibly erroneous messages (that is, messages whose membership status does not correspond to that of the sender) contained in the initial configuration. Moreover, by Lemma 8, at each round a process v sends its membership status $< \mathsf{m}_v >$ to all its neighbors, neighbors that will receive this message during the next round.

Now, consider a process v that changes its membership status during some round. The last time v changes its membership status during the round, it knows the real membership status of its neighbors, *i.e.*, $\forall u \in N(v)$, $\mathsf{m}_v[u] = \mathsf{m}_u$. So by changing its membership status, v solves its membership conflicts with its neighbors having a lower identity, and does not create any new conflict when joining the maximal independent set. Hence, in the worst case, processes stop changing their membership status after $O(n)$ rounds. In the worst case, at each round, a process replies to every message sent by its neighbors during the previous round. Thus, $O(\Delta kn^2)$ messages are exchanged in total. □

Again, to avoid starvation of one of the algorithms, we compose them using a *fair composition* [29], *i.e.*, each process alternatively executes a step of Algorithm 1 and a step of Algorithm 3. As demonstrated in previous work [17], fair composition preserves the property of self-stabilization.

6 Concluding Remarks

We presented the first deterministic self-stabilizing solution in asynchronous message-passing networks with links of unbounded capacity with unknown number of initial messages that requires only sub-logarithmic memory and message size (in n) bits, when Δ is itself sub-logarithmic.

Our approach is constructive and modular. In particular, our **DAG** algorithm layer solves a fundamental difficulty in many settings with respect to self-stabilization: avoiding cyclic behavior. We believe this can be a valuable asset when solving other problems in the same setting. We demonstrated the versatility of our **DAG** layer by providing two direct application algorithms: **Vertex coloring** and **Maximal Independent Set** construction.

Our **Vertex coloring** algorithm layer does not guarantee a locally minimal coloring (if an initial configuration is a coloring that is not locally minimal, no conflict is found and thus recoloring *does not* occurs). Nevertheless, a simple modification of the protocol following the lines of our **Maximal Independent Set** algorithm permits us to achieve this result: a node v that does not see a conflict but would like a smaller color asks its lower identity neighbors for authorization to take a new color c, any such neighbor u grants the authorization unless it has color c (but v didn't know it), or it itself wants to take a new color and is waiting for authorization. This mechanism does not create new conflicts, and any authorization chain length is bounded by the height of our constructed DAG. As a result, a minimal coloring is obtained.

We believe that **DAG**, **Vertex Coloring**, and **Maximal Independent Set** may prove useful for other local tasks, such as minimal dominating set, link coloring, or maximal matching. It would also be interesting to study their usefulness for solving global tasks (such as tree construction or leader election), and to see if the resource efficiency remains in this setting.

We would also like to mention two interesting open questions:

1. One property we retained about communication links is their FIFO behavior. The lifting of this hypothesis in the context of unbounded capacity links with unknown initial number of messages is likely to generate many impossibility results, which are left for future work.

2. The time complexity of the algorithms based on the **DAG**, *e.g.* **Vertex Coloring**, and **Maximal Independent Set** depends on the height of the DAG induced by the unique identifiers of the nodes. A possible way to reduce the DAG height it to use a self-stabilizing precoloring of the nodes (at some constant distance) so that the height is upper bounded by a constant. However, existing self-stabilizing solutions are randomized [28], and deterministic solutions [33] are not self-stabilizing and use the LOCAL model, where each node is capable of collecting all neighbors identifiers in a single synchronous step. Providing such a deterministic precoloring in our asynchronous model with constrained $O(\log \log n)$ bits messages while preserving self-stabilization is a promising challenge for future research.

References

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

[2] Yehuda Afek and Geoffrey M. Brown. Self-stabilization over unreliable communication media. *Distributed Computing*, 7(1):27–34, 1993.

[3] Ozkan Arapoglu, Vahid Khalilpour Akram, and Orhan Dagdeviren. An energy-efficient, self-stabilizing and distributed algorithm for maximal independent set construction in wireless sensor networks. *Comput. Stand. Interfaces*, 62:32–42, 2019.

[4] Leonid Barenboim. Deterministic $(\Delta + 1)$-coloring in sublinear (in Δ) time in static, dynamic, and faulty networks. *J. ACM*, 63(5):47:1–47:22, 2016. `doi:10.1145/2979675`.

[5] Leonid Barenboim, Michael Elkin, and Uri Goldenberg. Locally-iterative distributed $(\Delta + 1)$-coloring and applications. *Journal of the ACM*, 69(1):5:1–5:26, 2022.

[6] Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The locality of distributed symmetry breaking. In *53rd Annual IEEE Symposium on Foundations of Computer Science, FOCS 2012, New Brunswick, NJ, USA, October 20-23, 2012*, pages 321–330. IEEE Computer Society, 2012. doi:10.1109/FOCS.2012.60.

[7] Samuel Bernard, Stéphane Devismes, Katy Paroux, Maria Potop-Butucaru, and Sébastien Tixeuil. Probabilistic self-stabilizing vertex coloring in unidirectional anonymous networks. In *ICDCN'10*, volume 5935, pages 167–177, 2010.

[8] Samuel Bernard, Stéphane Devismes, Maria Gradinariu Potop-Butucaru, and Sébastien Tixeuil. Optimal deterministic self-stabilizing vertex coloring in unidirectional anonymous networks. In *IPDPS'09*, pages 1–8, Rome, Italy, 2009.

[9] Lélia Blin and Sébastien Tixeuil. Compact self-stabilizing leader election for general networks. In *LATIN'18*, volume 10807, pages 161–173, 2018.

[10] Janna Burman and Shay Kutten. Time optimal asynchronous self-stabilizing spanning tree. In Andrzej Pelc, editor, *Distributed Computing, 21st International Symposium, DISC 2007, Lemesos, Cyprus, September 24-26, 2007, Proceedings*, volume 4731 of *Lecture Notes in Computer Science*, pages 92–107. Springer, 2007. doi:10.1007/978-3-540-75142-7_10.

[11] James E. Burns, Mohamed G. Gouda, and Raymond E. Miller. Stabilization and pseudo-stabilization. *Distributed Computing*, 7(1):35–42, 1993.

[12] Sylvie Delaët, Bertrand Ducourthial, and Sébastien Tixeuil. Self-stabilization with r-operators revisited. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 3(10):498–514, 2006.

[13] Sylvie Delaët and Sébastien Tixeuil. Tolerating transient and intermittent failures. *Journal of Parallel and Distributed Computing (JPDC)*, 62(5):961–981, 2002.

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

[15] Shlomi Dolev. *Self-stabilization*. MIT Press, March 2000.

[16] Shlomi Dolev, Swan Dubois, Maria Potop-Butucaru, and Sébastien Tixeuil. Stabilizing data-link over non-FIFO channels with optimal fault-resilience. *Inf. Process. Lett.*, 111(18):912–920, 2011.

[17] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7(1):3–16, 1993.

[18] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Resource bounds for self-stabilizing message-driven protocols. *SIAM J. Comput.*, 26(1):273–290, 1997.

[19] Bertrand Ducourthial and Sébastien Tixeuil. Self-stabilization with r-operators. *Distributed Computing (DC)*, 14(3):147–162, 2001.

[20] Bertrand Ducourthial and Sébastien Tixeuil. Self-stabilization with path algebra. *Theoretical Computer Science (TCS)*, 293(1):219–236, February 2003.

[21] Mohsen Ghaffari. An improved distributed algorithm for maximal independent set. In Robert Krauthgamer, editor, *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 270–277. SIAM, 2016. doi:10.1137/1.9781611974331.ch20.

[22] Mohsen Ghaffari and Fabian Kuhn. Deterministic distributed vertex coloring: Simpler, faster, and without network decomposition. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*, pages 1009–1020. IEEE, 2021. doi:10.1109/FOCS52979.2021.00101.

[23] Sukumar Ghosh and Mehmet Hakan Karaata. A self-stabilizing algorithm for coloring planar graphs. *Distributed Computing*, 7(1):55–59, 1993.

[24] Wayne Goddard, Stephen T. Hedetniemi, David Pokrass Jacobs, and Pradip K. Srimani. Self-stabilizing protocols for maximal matching and maximal independent sets for ad hoc networks. In *17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France, CD-ROM/Abstracts Proceedings*, page 162. IEEE Computer Society, 2003.

[25] Mohamed G. Gouda and Nicholas J. Multari. Stabilizing communication protocols. *IEEE Trans. Computers*, 40(4):448–458, 1991. doi:10.1109/12.88464.

[26] Maria Gradinariu and Sébastien Tixeuil. Self-stabilizing vertex coloring of arbitrary graphs. In *OPODIS'00*, pages 55–70, 2000.

[27] Maria Gradinariu and Sébastien Tixeuil. Conflict managers for self-stabilization without fairness assumption. In *27th IEEE International Conference on Distributed Computing Systems (ICDCS 2007), June 25-29, 2007, Toronto, Ontario, Canada*, page 46. IEEE Computer Society, 2007. doi:10.1109/ICDCS.2007.95.

[28] Ted Herman and Sébastien Tixeuil. A distributed TDMA slot assignment algorithm for wireless sensor networks. In *ALGOSENSORS'04, First International Workshop on Algorithmic Aspects of Wireless Sensor Networks, Turku, Finland, July 16, 2004*, volume 3121, pages 45–58, 2004.

[29] Ted Richard Herman. *Adaptivity through distributed convergence.* PhD thesis, University of Texas, Austin, 1991.

[30] Michiyo Ikeda, Sayaka Kamei, and Hirotsugu Kakugawa. A space-optimal self-stabilizing algorithm for the maximal independent set problem. In *PDCAT'02 - Third International Conference on Parallel and Distributed Computing, Applications and Technologies, September 3-6, 2002, Kanazawa, Japan*, pages 70–74, 2002.

[31] Richard M. Karp and Avi Wigderson. A fast parallel algorithm for the maximal independent set problem. *Journal of the ACM*, 32(4):762–773, 1985.

[32] Shmuel Katz and Kenneth J. Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7(1):17–26, 1993.

[33] Nathan Linial. Legal coloring of graphs. *Comb.*, 6(1):49–54, 1986. doi:10.1007/BF02579408.

[34] Nathan Linial. Locality in distributed graph algorithms. *SIAM J. Comput.*, 21(1):193–201, 1992. doi:10.1137/0221015.

[35] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 15(4):1036–1053, 1986. doi:10.1137/0215074.

[36] Toshimitsu Masuzawa and Sébastien Tixeuil. On bootstrapping topology knowledge in anonymous networks. *ACM Transactions on Adaptive and Autonomous Systems (TAAS)*, 4(1), 2009.

[37] Nathalie Mitton, Bruno Séricola, Sébastien Tixeuil, Eric Fleury, and Isabelle Guérin-Lassous. Self-stabilization in self-organized multihop wireless networks. *Ad Hoc and Sensor Wireless Networks*, 11(1-2):1–34, 2011.

[38] Sumit Sur and Pradip K. Srimani. A self-stabilizing algorithm for coloring bipartite graphs. *Inf. Sci.*, 69(3):219–227, 1993.

[39] Sébastien Tixeuil. *Algorithms and Theory of Computation Handbook, Second Edition*, chapter Self-stabilizing Algorithms, pages 26.1–26.45. Chapman & Hall/CRC Applied Algorithms and Data Structures. CRC Press, Taylor & Francis Group, November 2009.

[40] Volker Turau. Linear self-stabilizing algorithms for the independent and dominating set problems using an unfair distributed scheduler. *Inf. Process. Lett.*, 103(3):88–93, 2007.

[41] George Varghese. Self-stabilization by counter flushing. *SIAM J. Comput.*, 30(2):486–510, 2000.

[42] George Varghese and Mahesh Jayaram. The fault span of crash failures. *J. ACM*, 47(2):244–293, 2000.