Optimal Space Lower Bound for Deterministic Self-Stabilizing Leader Election Algorithms

Lélia Blin
Sorbonne Université, Université d'Evry-Val-d'Essonne, CNRS, LIP6 UMR 7606, 4 place Jussieu, 75005 Paris, France

Laurent Feuilloley
Univ. Lyon, Université Lyon 1, LIRIS UMR CNRS 5205, F-69621, Lyon, France

Gabriel Le Bouder
Sorbonne Université, CNRS, INRIA, LIP6 UMR 7606, 4 place Jussieu, 75005 Paris, France

Abstract

Given a boolean predicate Π on labeled networks (e.g., proper coloring, leader election, etc.), a self-stabilizing algorithm for Π is a distributed algorithm that can start from any initial configuration of the network (i.e., every node has an arbitrary value assigned to each of its variables), and eventually converge to a configuration satisfying Π. It is known that leader election does not have a deterministic self-stabilizing algorithm using a constant-size register at each node, i.e., for some networks, some of their nodes must have registers whose sizes grow with the size n of the networks. On the other hand, it is also known that leader election can be solved by a deterministic self-stabilizing algorithm using registers of $O(\log \log n)$ bits per node in any n-node bounded-degree network. We show that this latter space complexity is optimal. Specifically, we prove that every deterministic self-stabilizing algorithm solving leader election must use $\Omega(\log \log n)$-bit per node registers in some n-node networks. In addition, we show that our lower bounds go beyond leader election, and apply to all problems that cannot be solved by anonymous algorithms.

2012 ACM Subject Classification Theory of computation → Distributed computing models

Keywords and phrases Space lower bound, memory tight bound, self-stabilization, leader election, anonymous, identifiers, state model, ring topology

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2021.24

Funding Support by ANR ESTATE (ANR-16-CE25-0009-03) and ANR GrR (ANR-18-CE40-0032).

Acknowledgements We thank the reviewers for their useful comments.

1 Introduction

1.1 Context

Self-stabilization is a paradigm suited to asynchronous distributed systems prone to transient failures. The occurrence of such a failure (e.g., memory corruption) may move the system to an arbitrary configuration. An algorithm is self-stabilizing if it guarantees that whenever the system is in a configuration that is illegal w.r.t. some given boolean predicate Π, the system returns to a legal configuration in finite time (and remains in legal configuration as long as no other failures occur). In this paper, we study self-stabilization in networks. The network is modeled as a graph, and we consider predicates defined on labeled graphs. For instance, in proper k-coloring, a configuration is legal if every node is labeled by a color $\{1, \ldots, k\}$ that is different from the colors of all its neighbors. Given a boolean predicate Π, a self-stabilizing algorithm for Π is a distributed algorithm enabling every node, given any input state, to construct a label such that the resulting labeled graph satisfies Π.

During the execution of a self-stabilizing algorithm, the nodes exchange information along the links of the network, and this information is stored locally at every node. Specifically, processes in a distributed system have two types of memory: the *persistent* memory, and the *mutable* memory. The persistent memory is used to store the identity of the process (e.g., its globally unique MAC address), its port numbers, and the code of the algorithm executed on the process. Importantly, this section of the memory is not write enabled during the execution of the algorithm. As a consequence it is less likely to be corruptible, and most work in self-stabilization assumes that this part of the memory is not subject to failures. The mutable memory is used to store the variables used by the algorithm, and is subject to failures, that is, to the corruption of these variables. The space complexity of a self-stabilizing algorithm is the total size of all the variables used by the algorithm, including those used to encode the output label of the node. For instance, the space complexity of the algorithm for k-coloring is at least $\Omega(\log k)$ bits per node, for encoding the colors in $\{1, \ldots, k\}$. The question addressed in this paper is: under which circumstances is it possible to reach a space complexity as low as the size of the labels? And if not, what is the smallest space complexity that can be achieved?

Preserving small space complexity is indeed very much desirable, for several reasons. First, it is expected that self-stabilizing algorithms offer some form of universality, in the sense that they are executable on several types of networks. Networks of sensors as used in IoT, as well as networks of robots as used in swarm robotics, have the property to involve nodes with limited memory capacity, and distributed algorithms of large space complexity may not be executable on these types of networks. Second, a small space complexity is the guarantee to consume a small bandwidth when nodes exchange information, thus reducing the overhead due to link congestion [1]. In fact, a self-stabilizing algorithm is never terminating, in the sense that it keeps running in the background in case a failure occurs, for helping the system to return to a legal configuration. Therefore, nodes may be perpetually exchanging information, even after stabilization, and even when no faults occur. Limiting the amount exchanged information, and thus, in particular, the size of the variables, is therefore of the utmost importance for optimizing time, and even energy. Last but not least, increasing robustness against variable corruption can be achieved by data replication [17]. This is however doable only if the variables are reasonably small. Said otherwise, for a given memory capacity, the smaller the space complexity the larger the robustness thanks to data replication.

1.2 Contributions of the paper

In this paper, we focus on one of the arguably most important problems in the context of distributed computing, namely *leader election*. The objective is to maintain a unique leader in the network, and to enable the network to return to a configuration with a unique leader in case there are either zero or more than one leaders. Interestingly, encoding legal states consumes one single bit at each node. Indeed, in leader election, every node has a label with value 0 or 1, and these labels form a legal configuration if there is one and only one node with label 1. As a consequence, up to an additive constant, the space complexity is exactly the space used to encode the variables of the algorithm (which is not the case for other problems where the output itself uses some non-trivial space to be encoded).

We establish the lower bound of $\Omega(\log \log n)$ bits per node for the space complexity of leader election. This improves the only lower bound known so far (see [3]), which states that leader election has non-constant space complexity, i.e., complexity $\omega(1)$, where $f = \omega(g)$ if $g(n)/f(n) \to 0$ when $n \to \infty$. More importantly, our bound matches the best known upper

bound on the space complexity of leader election, which is $O(\log \log n)$ bits per node in bounded degree networks [6], and in particular invalidates the folklore conjecture stating that leader election is solvable using only $O(\log^* n)$ bits of mutable memory per node.

We obtain our lower bound by establishing an interesting connection between self-stabilizing algorithms with small space complexity, and self-stabilizing algorithms performing in anonymous networks, that is, in networks in which nodes have no identifiers. (Recall that space complexity counts solely the size of the mutable memory, and does not include the immutable persistent memory where the identifiers are stored). More specifically, the technical ingredient used for establishing our results are the following. It is known that many self-stabilization problems, including vertex coloring, leader election, spanning tree construction, etc., require that the nodes are provided with identifiers, for breaking symmetry. Indeed, no algorithm can solve these problems in anonymous networks (under a standard distributed scheduler). We show that, for any self-stabilizing algorithm in a network with node identifiers, if the space complexity of the algorithm is too small, then the algorithm does not have more power than a self-stabilizing algorithm running in an anonymous network. More precisely, let A be an algorithm in a network with node identifiers, and let us assume that A has space complexity $o(\log \log n)$ bits per node. Such a small space complexity does not prevent A from exchanging identifiers between nodes, but they must be transferred as a series of smaller pieces of information that are pipelined along a link, each of size $o(\log \log n)$ bits. On the other hand, a node cannot store the identifier of even just one of its neighbors. We show that, with spacial complexity $o(\log \log n)$ bits per node, there exist graphs and assignments of identifiers to the nodes of these graphs such that, in these graphs and for these assignments, A has the same behavior as an algorithm executed in these graphs but in the absence of identifiers (i.e., in the anonymous version of these graphs). We then show that no algorithms can solve leader election in these graphs in absence of identifiers, from which it follows that A cannot solve leader election in these graphs with identifiers as long as its space complexity is $o(\log \log n)$ bits per node.

1.3 Related work

Space complexity of self-stabilizing algorithms has been extensively studied for *silent* algorithms, that is, algorithms that guarantee that the content of the variables of every node does not change once the algorithm has reached a legal configuration. For silent algorithms, Dolev and al. [11], proved that finding the centers of a graph, electing a leader, and constructing a spanning tree require registers of $\Omega(\log n)$ bits per node. Silent algorithms have later been related to a concept known as *proof-labeling scheme* (PLS) [23]. Any lower bound on the size of the proofs in a PLS for a predicate Π on labeled graph implies a lower bound on the size of the registers for silent self-stabilizing algorithms solving Π. A typical example is the $\Omega(\log^2 n)$-bit lower bound on the size of any PLS for minimum-weight spanning trees (MST) [22], which implies the same bound for constructing an MST in a silent self-stabilizing manner [4]. Thanks to the tight connection between silent self-stabilizing algorithms and proof-labeling schemes, the space complexity of a vast collection of problems is known, for silent algorithms. (See [14] for more information on proof-labeling schemes.)

On the other hand, to our knowledge, the only lower bound on the space complexity of problems for general self-stabilizing algorithms (without the requirement of being silent) that is corresponding to our setting has been established by Beauquier et al. [3] who proved that registers of constant size are not sufficient for leader election algorithms. Interestingly, the same paper also contains several other space complexity lower bounds for models different from ours – e.g., anonymous networks, or harsher form of asynchrony.

The literature dealing with upper bounds is far richer. In particular, [5] recently presented a self-stabilizing leader election algorithm using registers of $O(\log\log n)$ bits per node in n-node rings. This algorithm was later generalized to networks with maximum degree Δ, using registers of $O(\log\log n + \log\Delta)$ bits per node [6]. It is worth to notice that spanning tree construction and $(\Delta+1)$-coloring have the same space complexity $O(\log\log n + \log\Delta)$ bits per node [6]. Prior to these work, the best upper bound was a space complexity $O(\log n)$ bits per node [5], and it has then been conjectured that, by some iteration of the technique enabling to reduce the space complexity from $O(\log n)$ bits per node to $O(\log\log n)$ bits per node, one could go all the way down to a space complexity of $O(\log^* n)$ bits per node. Arguments in favor of this conjecture were that such successive exponential improvements have been observed several times in distributed computing. A prominent example is the time complexity of minimum spanning tree construction in the congested clique model [24, 16, 15, 21]. Complexities $O(\log^* n)$ do exist in the self-stabilizing framework [2], and it seemed at first that the technique in [11] could indeed be iterated (in a similar fashion as in [7]). Our results shows that this is not the case, and that $\Theta(\log\log n)$ is the right answer.

2 Model and definitions

In this paper, we are considering the *state model* for self-stabilization [9]. The asynchronous network is modeled as a simple n-nodes graph $G = (V, E)$, where the set of the nodes V represents the processes, and the set of edges E represents pairs of processes that can communicate directly with each other. Such pairs of processes are called *neighbors*. The set of the neighbors of node v is denoted $N(v)$. Each node has local variables and a local algorithm. The variables of a node are stored in its mutable memory, also called register. In the state model, each node v has read/write access to its register. Moreover, in one atomic step, every node reads its own register and the registers of its neighbors, executes its local algorithm and updates its own register if necessary. Note that the values of the variables of one node $v \in V$ are called the *state* of v, and denoted by $S(v)$.

Each node $v \in V$ has a distinct identity, denoted by $ID(v) \in \{1, \ldots, n^c\}$ for some constant $c > 1$. For each adjacent edge, each node has access to a locally unique port number. No assumption is made on the consistency between port numbers on each node. The mutable memory is the memory used to store the variables, while the immutable memory is used to store the identifier, the port numbers, and the code of the protocol. As a consequence, the identity and the port numbers are non corruptible constants, and only the mutable memory is considered when computing the memory complexity because it corresponds to the memory readable by the neighbors of the nodes, and thus correspond to the information transmitted during the computation. More precisely, an algorithm may refer to the identity, or to the port numbers, of the node, without the need to store them in the variables. If at least one rule of an algorithm refers to the identity of the node, we call this algorithm an *ID-based algorithm*. Otherwise, if the rules do not refer to the identity of the node we say it is an *anonymous algorithm*.

The output of the algorithm for a problem is carried through local variables of each node. The output of the problem may use all the local variables, or only a subset of them. Indeed, the algorithm must have local variables that match the output of the problem, we call these variables the specification variables. But the algorithm may also need some extra local variables, that may be necessary to compute the specification variables. For example, if we consider a silent BFS spanning tree construction, the specification variables are the variables

dedicated to pointing out the parent in the BFS. However, to respect the silent property, the algorithm needs in each node a variable dedicated to the identity of the root of the spanning tree and a variable dedicated to the distance from the root. As a consequence, we define the *specification* of problem P as a description of the correct assignments of specification variables, for this specific problem P.

A *configuration* is an assignment of values to all variables in the system, let us denote by Γ the set of all the configurations. A *legal configuration* is a configuration γ in Γ that respects the specification of the problem, we denote by Γ^* the set of legal configurations. A local algorithm is a set of rules the node can apply, each rule is of the form *<label>:<guard>→<command>*. A *guard* is a Boolean predicate that uses the local variables of the node and of its neighbors, and a *command* is an assignment of variables. A node is said to be *enabled* if one of its guard is true and *disabled* otherwise.

We consider an asynchronous network, the asynchrony of the system is modeled by an adversary called *scheduler* or *daemon*. The scheduler chooses, at each step, which enabled nodes will execute a rule. Several schedulers are proposed on the literature depending on their characteristics. Dubois and al. in [12] presented a complete overview of these schedulers. Since we are interested in showing a lower bound, we aim for the least challenging scheduler. Our lower bound is established under the *synchronous distributed scheduler*, a strongly fair distributed scheduler. The synchronous distributed scheduler activates, at each step, all the enabled nodes. The synchronous distributed scheduler is captured by the weakly fair and unfair distributed schedulers, but not by the central scheduler that activates only one at each step.

A configuration $\gamma \in \Gamma$ is a legal configuration for the leader election problem if one single node is elected. More formally:

▶ **Definition 1** (Leader election). *Leader election in $G = (V, E)$ is specified by a boolean variable ℓ_v at each node $v \in V$. A configuration $\{(v, \ell_v) : v \in V\}$ is legal if there is a node $v \in V$ such that $\ell_v = true$, and for every other node $u \in V \setminus \{v\}$, $\ell_u = false$.*

3 Formal Statement of the Results

3.1 Lower bounds

Our first result is an $\Omega(\log \log n)$ lower bound for leader election on the cycle.

▶ **Theorem 2.** *Let $c > 1$. Every deterministic self-stabilizing algorithm solving leader election in the state model under a strongly fair distributed scheduler requires registers on $\Omega(\log \log n)$ bits per node in n-node graphs with unique identifiers in $[1, n^c]$.*

This bound improves the only lower bound known so far [3], from $\Omega(1)$ to $\Omega(\log \log n)$, and it is tight, as it matches the upper bound of [5]. In particular, it invalidates the folklore conjecture stating that the aforementioned problems are solvable using only $O(\log^* n)$ memory.

Optimality of the assumptions

Our lower bound is actually optimal not only in term of size, but also in terms of the assumptions we make on the setting. More precisely, our theorem has three restrictions: it works for deterministic algorithms only, with the distributed scheduler, and with identifiers in a large enough range. We will now discuss why these limitations are actually necessary.

Randomization is a common tool for symmetry breaking, and our problem is one example. Namely, [18] proved that using randomization, one can solve leader election using constant memory, which implies that our result cannot be generalized in that direction.

An important aspect of the self-stabilizing setting is the scheduler, which is the adversary that decides which nodes can take a step at each round. Different schedulers model different assumptions on the asynchrony of the setting. For example, a fully adversarial scheduler can take any decision, as long as at least one node can take a step at each round. Weaker schedulers can delay arbitrarily the step of a node but are forced to eventually activate any node, etc. Our lower bound is valid under a very weak scheduler, which means that we need a weak assumption on the asynchrony, which in turn means that our result is strong on this aspect. Precisely, what we need in our proof is that it is possible for the scheduler to activate all the nodes at every round. One type of scheduler for which this property does not hold is the so-called centralized scheduler, that activates exactly one node at every round. In this context the symmetry is broken by the scheduler itself (if two nodes are in the same situation, one will be activated first, and this breaks the symmetry). Although the proof of Theorem 2 in Section 4 does not apply to the centralized scheduler, the more genereal Theorem 3 will allow us to extend our result to the central daemon.

Finally, and this is probably more surprising, we need to consider identities between 1 and n^c, for $c > 1$. In particular, our technique does not work if the identifiers are in $O(n)$. This is not an artifact of our proof: it is actually necessary for the result to hold. Indeed, if the identifier range is $[1, n]$, then an algorithm may use the node with identifier 1 as a designated node, and have a special code for it. Algorithms using such designated nodes are called *semi-uniform algorithms* and they can achieve space complexity below our lower bound [8, 20]. Even without the possibility of having a designated node, one can take advantage of smaller identifier range: there actually exists an algorithm in constant memory if the identities are in $[1, n + k]$ for a constant k [3].

A general result on the power of the identifiers

Actually, our technique goes beyond the setting of Theorem 2. First, we do not need the harshest aspects of the self-stabilizing model which is that the initial configuration can be arbitrary. If we start from an empty configuration, our technique still holds. Second, the technique works for basically any problem that requires minimal symmetry breaking, not just leader election. Third, as hinted above the type of scheduler is not really important, as long as it does not break symmetry. We prove the more general following theorem.

▶ **Theorem 3.** *Let $c > 1$, and let $\Delta(n) \in o(\log n)$ be a function. If there exists a deterministic self-stabilizing algorithm $\mathcal{A}$ that solves a problem $\mathcal{P}$ in the state model and uses registers of size $o(\frac{\log\log n}{\Delta})$, then there exists a deterministic self-stabilizing anonymous algorithm $\mathcal{A}^a$ that solves $\mathcal{P}$ on every large-enough graph with maximum degree $\Delta(n)$.*

What our paper is really about, is the power of identifiers, in a scenario where very little space/communication is used. Our core result is that below $\Theta(\log\log n)$, identifiers are useless, in the sense that in the worst-case the performance of an algorithm using these identifiers is the same as the performance of an anonymous algorithm. Remember that the Naor-Stockmeyer order-invariance theorem [25], that states that in the LOCAL model, for local problems, constant-time algorithms that use the exact values of the identifiers are not more powerful than the order-invariant algorithm that only use the relative ordering of the identifiers. In some sense our paper and [25] have the same take-home message, in two different contexts: if you do not have enough resources, you cannot use the (full) power of the identifiers.

Theorem 3 establishes that proving a $\Omega(\log \log n)$ lower bounds in the ID-based setting boils down to proving that anonymous algorithms cannot solve the problem. This is useful, because indistinguishability arguments are easier to establish for anonymous algorithms than for algorithms using identifiers.

Finally, one aspect that is not explicit in the statement of the theorem but follows from the proof, is that actually this result also holds if the nodes have inputs. In particular, our result applies to the semi-uniform setting where exactly one node has a special input.

About the port number model

Our general theorem, Theorem 3, holds in the model where a node knows its port-numbers but not the ones of its neighbors. Intuitively, this implies that a node u cannot specify that some piece of information is intended to the node of port number p, because that node does not know it has been assigned port-number p.

In some graphs, the port number assignment can be chosen in such a way that knowing the port-number assignment of the neighbors does not help. For example in the cycle of Theorem 2, we can arrange the port numbers such that every edge is assigned port number 1 by one endpoint, and port number 2 by the other. This allows to generalize our first result to the model where a node knows both port numbers on every adjacent edge.

Central scheduler

In a celebrated paper [10], Dijkstra established, among other things, that one cannot break symmetry with anonymous algorithms in composite rings (that is, in rings whose size n is not a prime number). This results holds under a *central* strongly fair scheduler. A central scheduler is somehow the opposite of the distributed scheduler used in Theorem 2: it activates one node at every round instead of activating all of them. Yet, we can generalize Dijkstra's result. Indeed, as highlighted before, the core of our proofs, and the statement of Theorem 3, is about proving that an algorithm with too little memory cannot perform better than an anonymous algorithm, and this does not depend on the scheduler. Therefore, by combining Dijkstra's result and Theorem 3, we directly get the following corollary, that complements Theorem 2.

▶ **Corollary 4.** *Let $c > 1$. Every deterministic self-stabilizing algorithm solving leader election in the state model under a strongly fair* central scheduler *requires registers on $\Omega(\log \log n)$ bits per node in n-node composite rings with unique identifiers in $[1, n^c]$.*

Note that assuming that the ring is composite is essential, since [19] builds a constant memory algorithm for leader election on anonymous prime rings, under a central scheduler.

3.2 Intuition of the proofs

Challenge of lower bounds for non-silent algorithms

Almost all lower bounds for self-stabilization are for silent algorithms, that are required to stay in the same configuration once they have stabilized. These lower bounds are then about a static data structure, the stabilized solution. The question boils down to establishing how much memory is needed to locally certify the global correctness of the solution, and this is well-studied [13].

When we do not require that the algorithm should converge to one correct configuration, and stay there, there is no static structure on which we can reason. It is then unclear how we can establish lower bounds. One way is to think about invariants. Consider a property that

we can assume to hold in the initial configuration, and that is preserved by the computation (if it follows some memory requirement hypothesis). If no correct output configuration has this property, then we can never reach a correct output configuration.

In our proof, the property that will be preserved is that every node has the same state. This can clearly be assumed for the original configuration, and we show that basically if the memory is limited then this is preserved at each step. As the specification we use for leader election is that the leader should output 1, and the other nodes should output 0, then it is not possible that all nodes have the same state in a proper output configuration.

Intuition on a toy problem

Let us now give some intuition about why we can replace ID-based algorithms by anonymous ones. The code of an ID-based algorithm $\mathcal{A}$ may refer to the identifier of the node that is running it. For example, a rule of the algorithm could be:

- if the states of the current node and of its left and right neighbors are respectively x, y, and z, then: if the identifier is odd the new state is a, otherwise it is b.

Now suppose you have fixed an identifier, and you look at the rules for this fixed identifier. In our example, if the identifier is 7, the rule becomes:

- if the states of the current node and of its left and right neighbors are respectively x, y, and z, then: the new state is a.

This transformation can be done for any rule, thus, for an identifier i, we can get an algorithm $\mathcal{A}_i$ specific to this identifier. When we run $\mathcal{A}$ on every node, we can consider that every node, with some identifier i is running $\mathcal{A}_i$. Note that $\mathcal{A}_i$ does not refer in its code to the identifier.

The key observation is the following. If the amount of memory an algorithm can use is very limited, then there is very limited number of different behaviors a node can have, especially if the code does not refer to the identifier. Let us illustrate this point by studying an extreme example: a ring on which states have only one bit. In this case the number of input configurations for a node, is the set of views (x, y, z) as above, with $x, y, z \in \{0, 1\}$. That is there are $2^3 = 8$ different inputs, thus the algorithm can be described with 8 different rules. Since the output of the function is the new state, the output is also a single bit. Therefore, there are at most $2^8 = 256$ different sets of rules, that is 256 different possible behaviors for a node. In other words, in this extreme case, each specific algorithm $\mathcal{A}_i$ is equal to one of the behaviors of this list of 256 elements. This implies that, if we take a ring with 257 nodes, there exist two nodes with two distinct identifiers i and j, such that the specific algorithms $\mathcal{A}_i$ and $\mathcal{A}_j$ are equal.

This toy example is not strong enough for our purpose, as we want to argue about instances where all the nodes run the same code, and as we want non-constant memory. But the idea above can be strengthened to get our theorem. The key is to use the hypothesis that the identifiers are taken from a polynomially large range. As we have a pretty large palette of identifiers, we can always find, not only 2, but n distinct identifiers in $[1, n^c]$, such that all the specific algorithms $\mathcal{A}_i$ correspond to the exact same behavior. In this case it is as if the algorithm were anonymous.

Note that the larger the memory is, the more different behaviors there are, and the smaller the set of identical specific algorithms we can find. This trade-off implies that for polynomial range, the construction works as long as the memory is in $o(\log \log n)$.

4 Proof of Theorem 2

Consider a ring of size n, and an ID-based algorithm $\mathcal{A}$ using $f(n)$ bits of memory per node to solve leader election. An algorithm can be seen as the function that describes the behavior of the algorithm. This function takes an identifier, a state for the node, a state for its left neighbor and a state for its right neighbor, and gives the new state of the node. Formally:

$$\begin{array}{rccccccccc} \mathcal{A}: & [n^c] & \times & \{0,1\}^{f(n)} & \times & \{0,1\}^{f(n)} & \times & \{0,1\}^{f(n)} & \to & \{0,1\}^{f(n)} \\ & (ID & , & \text{state} & , & \text{left-state} & , & \text{right-state}) & \mapsto & \text{new-state} \end{array}$$

Note that in general, we consider non-directed rings thus the nodes do not have a global consistent definition for right and left. As we are dealing with a lower bound with a worst-case on the port numbering, assuming such a consistent orientation only makes the result stronger. Now we can consider that for every identifier i, we have an algorithm of the form:

$$\begin{array}{rccccccc} \mathcal{A}_i: & \{0,1\}^{f(n)} & \times & \{0,1\}^{f(n)} & \times & \{0,1\}^{f(n)} & \to & \{0,1\}^{f(n)} \\ & (\text{state} & , & \text{left-state} & , & \text{right-state}) & \mapsto & \text{new-state} \end{array}$$

Thus a specific algorithm $\mathcal{A}_i$ boils down to a function of the form: $\{0,1\}^{3f(n)} \to \{0,1\}^{f(n)}$. Let us call such a function a *behaviour*, and let $\mathcal{B}_n$ be the sets of all behaviours.

▶ **Lemma 5.** $|\mathcal{B}_n| = 2^{f(n)\times 2^{3f(n)}}$

Proof. The inputs are basically binary strings of length $3f(n)$, thus there are $2^{3f(n)}$ possibilities for them. Similarly the number of possible outputs is $2^{f(n)}$. Thus the number of functions in $|\mathcal{B}_n|$ is $\left(2^{f(n)}\right)^{2^{3f(n)}} = 2^{f(n)\times 2^{3f(n)}}$. ◀

Lemma 5 implies that the smaller f, the fewer different behaviors. Let us make this more concrete with Lemma 6.

▶ **Lemma 6.** *If $f(n) \in o(\log\log n)$, then for every n large enough, $n^{c-1} > |\mathcal{B}_n|$.*

Proof. Consider the expression of n^{c-1} and $|\mathcal{B}_n|$ after applying the logarithm twice:

$$\begin{aligned} \log\log(n^{c-1}) &= \log(c-1) + \log\log n \\ &\sim \log\log n \end{aligned}$$

$$\begin{aligned} \log\log(|\mathcal{B}_n|) &= \log\log\left(2^{f(n)\times 2^{3f(n)}}\right) = \log\left(f(n)\times 2^{3f(n)}\right) = \log(f(n)) + 3f(n) \\ &\sim 3f(n) \end{aligned}$$

As the dominating term in the second expression is of order $f(n) \in o(\log\log n)$, asymptotically the first expression is larger. As $\log\log(\cdot)$ is an increasing positive function for large values, this implies that asymptotically $n^{c-1} > |\mathcal{B}_n|$. ◀

The next lemma shows that if $f(n) \in o(\log\log n)$, we can find a large number of identifiers that have the same specific algorithm.

▶ **Lemma 7.** *If $n^{c-1} > |\mathcal{B}_n|$, then there exist a behavior function b, and a set S of n different identifiers, such that: $\forall i \in S$, $\mathcal{A}_i = b$.*

Proof. Let the function $\varphi : [n^c] \to \mathcal{B}_n$ be the function that associates each identifier to its corresponding behavior function. Let S be the function that associates to each behavior b, the set of identifiers i such that $\varphi(i) = b$. Since φ is a function from $[n^c]$ to $\mathcal{B}_n$, its inverse function S satisfies: $\cup_{b\in\mathcal{B}_n} S(b) = [n^c]$. Thereby, the average size of a set $S(b)$ is

$$\frac{1}{|\mathcal{B}_n|}\sum_{b\in\mathcal{B}_n}|S(b)| = \frac{1}{|\mathcal{B}_n|}\cdot n^c.$$

Because of Lemma 6, this quantity is strictly larger than n. Thus, the average of $|S(b)|$ among all b is larger than n, and there must exist at least one behavior b such that $|S(b)| > n$. We take this b and $S = S(b)$ for the lemma. ◀

Combining the three lemmas we get that, if $f(n) \in o(\log\log n)$, then for any large enough n we can find n different identifiers in $[1, n^c]$, such that the nodes have the exact same behavior.

Now, consider a large enough graph, with identifiers taken from the set S. As we are in a self-stabilizing scenario, we can chose from which configuration we start. We actually do not need intricate configuration: we start from a configuration where all states are the same, say some string x. This cannot be a proper leader election output, because in leader election (with our specification) a leader and a non leader have different output. Therefore, at least one node is activable. Note that every node sees the same states x for itself, and its two neighbors, and that by construction they have the same behavior. Therefore if one node is activated, all nodes are activated. Now, the scheduler decides to activate all the nodes. Necessarily, all the nodes take the exact same step, and end up in a configuration where every node has the same state.

We can iterate this argument forever. In other other words, the network cannot escape the set of configurations where all nodes have the same state (as long as the scheduler activates all the nodes together, which is allowed). Therefore, the network will never reach a proper leader election configuration configuration, and this proves Theorem 2.

5 Proof of Theorem 3

In this section, we prove our general result, that relates ID-based algorithms with small memory to anonymous algorithms. The proof of Theorem 3 follows from the same idea as the one of Theorem 2, but in a more general way. (The constants of the statement have not been optimized.) Consider an algorithm $\mathcal{A}$ that solves a problem $\mathcal{P}$ in the state model and uses registers of size $o(\frac{\log\log n}{\Delta})$ (where the Δ is the maximum degree of the graphs treated by $\mathcal{A}$, and is bounded as a function of n by $\Delta(n) \in o(\log n)$).

As before an algorithm can be seen as a function that maps the view of the node to an output. But now, since the degree is not the same for every node, we would have to consider a different function for each degree $d \in [\Delta]$. This not very convenient for us, so let us take another point of view. We take a unique function, that takes as input: an identifier, a state, an integer δ that represents the degree degree of the vertex at hand, and Δ states.

$$\begin{array}{lccccccccc} \mathcal{A}: & [n^c] & \times & \{0,1\}^{f(n)} & \times & [\Delta] & \times & \left(\{0,1\}^{f(n)}\right)^{\Delta} & \to & \{0,1\}^{f(n)} \\ & (ID & , & \text{state} & , & \text{degree} & , & \Delta \text{ states}) & \mapsto & \text{new-state} \end{array}$$

To compute the output of a node with degree $\delta < \Delta$, the function simply ignores the last $(\Delta - \delta)$ inputs.

Note that this corresponds to the single-port setting: in the function, each neighboring state is identified. In particular, it is not a set of neighboring states. But it is not a double-port setting: which port a neighbor assigns to a node is unknown.

Now if we fix the identifier i, we get:

$$\begin{array}{rccccccc} \mathcal{A}_i: & \{0,1\}^{f(n)} & \times & [\Delta] & \times & \left(\{0,1\}^{f(n)}\right)^{\Delta} & \to & \{0,1\}^{f(n)} \\ & (\text{state} & , & \text{degree} & , & \Delta \text{ states}) & \mapsto & \text{new-state} \end{array}$$

Now the equivalent of Lemma 5 is that the number of such behaviors $\mathcal{A}_i$ is:

$$|\mathcal{B}_n| \leq \left(2^{f(n)}\right)^{2^{(\Delta+1)f(n)}\Delta}$$

We bound the double logarithm of this expression:

$$\begin{aligned} \log\log|\mathcal{B}_n| &\leq \log\log\left[\left(2^{f(n)}\right)^{2^{(\Delta+1)f(n)}\Delta}\right] \\ &\leq \log\left[f(n)2^{(\Delta+1)f(n)}\Delta\right] \\ &\leq (\Delta+1)f(n) + \log f(n) + \log\Delta(n) \\ &\in o(\log\log n) \end{aligned}$$

We get that for any large enough n, $n^{c-1} > |\mathcal{B}_n|$, and by Lemma 7, we directly get that there exists an identifier assignment such that all nodes have the same behavior. In other words, on this identifier assignment, the algorithm behaves like anonymous. As a consequence any impossibility result for anonymous algorithms also holds here. This completes the proof of Theorem 3.

6 Conclusion

In this paper, we have established a lower bound $\Omega(\log\log n)$ bits per node on the size of the registers for self-stabilizing algorithms solving leader election in the state model. This bound matches the upper bound $O(\log\log n)$ bits per node for bounded-degree graphs [6]. The same upper bound on the size of the registers is known to hold in bounded-degree graphs for vertex coloring and spanning tree construction [6]. An interesting open problem is to determine the space complexity of vertex coloring and spanning tree construction.

References

1. Jordan Adamek, Mikhail Nesterenko, and Sébastien Tixeuil. Evaluating practical tolerance properties of stabilizing programs through simulation: The case of propagation of information with feedback. In *Stabilization, Safety, and Security of Distributed Systems - 14th International Symposium, SSS 2012*, pages 126–132, 2012. doi:10.1007/978-3-642-33536-5_13.
2. Baruch Awerbuch and Rafail Ostrovsky. Memory-efficient and self-stabilizing network RESET (extended abstract). In *13th Annual ACM Symposium on Principles of Distributed Computing, PODC 1994*, pages 254–263, 1994. doi:10.1145/197917.198104.
3. Joffroy Beauquier, Maria Gradinariu, and Colette Johnen. Memory space requirements for self-stabilizing leader election protocols. In *18th Annual ACM Symposium on Principles of Distributed Computing, PODC 1999*, pages 199–207, 1999. doi:10.1145/301308.301358.
4. Lélia Blin and Pierre Fraigniaud. Space-optimal time-efficient silent self-stabilizing constructions of constrained spanning trees. In *35th IEEE International Conference on Distributed Computing Systems, ICDCS 2015*, pages 589–598, 2015. doi:10.1109/ICDCS.2015.66.
5. Lélia Blin and Sébastien Tixeuil. Compact deterministic self-stabilizing leader election on a ring: the exponential advantage of being talkative. *Distributed Computing*, 31(2):139–166, 2018. doi:10.1007/s00446-017-0294-2.

6 Lélia Blin and Sébastien Tixeuil. Compact self-stabilizing leader election for general networks. In *LATIN 2018: Theoretical Informatics - 13th Latin American Symposium*, pages 161–173, 2018. doi:10.1007/978-3-319-77404-6_13.

7 Lucas Boczkowski, Amos Korman, and Emanuele Natale. Minimizing message size in stochastic communication patterns: Fast self-stabilizing protocols with 3 bits. In *28th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017*, pages 2540–2559, 2017. doi:10.1137/1.9781611974782.168.

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

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

10 Edsger W. Dijkstra. *Self-Stabilization in Spite of Distributed Control*, pages 41–46. Springer New York, New York, NY, 1982. doi:10.1007/978-1-4612-5695-3_7.

11 Shlomi Dolev, Mohamed G. Gouda, and Marco Schneider. Memory requirements for silent stabilization. *Acta Inf.*, 36(6):447–462, 1999. doi:10.1007/s002360050180.

12 Swan Dubois and Sébastien Tixeuil. A taxonomy of daemons in self-stabilization, 2011. arXiv:1110.0334.

13 Laurent Feuilloley. Introduction to local certification. *CoRR*, abs/1910.12747, 2019. arXiv:1910.12747.

14 Laurent Feuilloley and Pierre Fraigniaud. Survey of distributed decision. *Bulletin of the EATCS*, 119, 2016. URL: http://bulletin.eatcs.org/index.php/beatcs/article/view/411/391, arXiv:1606.04434.

15 Mohsen Ghaffari and Merav Parter. MST in log-star rounds of congested clique. In *2016 ACM Symposium on Principles of Distributed Computing, PODC 2016*, pages 19–28, 2016. doi:10.1145/2933057.2933103.

16 James W. Hegeman, Gopal Pandurangan, Sriram V. Pemmaraju, Vivek B. Sardeshmukh, and Michele Scquizzato. Toward optimal bounds in the congested clique: Graph connectivity and MST. In *2015 ACM Symposium on Principles of Distributed Computing, PODC 2015*, pages 91–100, 2015. doi:10.1145/2767386.2767434.

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

18 Gene Itkis and Leonid A. Levin. Fast and lean self-stabilizing asynchronous protocols. In *35th Annual Symposium on Foundations of Computer Science FOCS 1994*, pages 226–239, 1994. doi:10.1109/SFCS.1994.365691.

19 Gene Itkis, Chengdian Lin, and Janos Simon. Deterministic, constant space, self-stabilizing leader election on uniform rings. In *Distributed Algorithms, 9th International Workshop, WDAG '95*, pages 288–302, 1995. doi:10.1007/BFb0022154.

20 Colette Johnen. Memory efficient, self-stabilizing algorithm to construct BFS spanning trees. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing, PODC 97*, page 288, 1997. doi:10.1145/259380.259508.

21 Tomasz Jurdzinski and Krzysztof Nowicki. MST in $O(1)$ rounds of congested clique. In *29th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018*, pages 2620–2632, 2018. doi:10.1137/1.9781611975031.167.

22 Amos Korman and Shay Kutten. Distributed verification of minimum spanning trees. *Distributed Computing*, 20(4):253–266, 2007. doi:10.1007/s00446-007-0025-1.

23 Amos Korman, Shay Kutten, and David Peleg. Proof labeling schemes. *Distributed Computing*, 22(4):215–233, 2010. doi:10.1007/s00446-010-0095-3.

24 Zvi Lotker, Boaz Patt-Shamir, Elan Pavlov, and David Peleg. Minimum-weight spanning tree construction in $O(\log \log n)$ communication rounds. *SIAM J. Comput.*, 35(1):120–131, 2005. doi:10.1137/S0097539704441848.

25 Moni Naor and Larry J. Stockmeyer. What can be computed locally? *SIAM J. Comput.*, 24(6):1259–1277, 1995. doi:10.1137/S0097539793254571.