Silent MST approximation for tiny memory

Lélia Blin
LIP6, Sorbonnes University
lelia.blin@lip6.fr

Swan Dubois
LIP6, Sorbonnes University
swan.dubois@lip6.fr

Laurent Feuilloley
University of Chile
feuilloley@dii.uchile.cl

Abstract

In network distributed computing, minimum spanning tree (MST) is one of the key problems, and silent self-stabilization one of the most demanding fault-tolerance properties. For this problem and this model, a polynomial-time algorithm with $O(\log^2 n)$ memory is known for the state model. This is memory optimal for weights in the classic $[1, \text{poly}(n)]$ range (where n is the size of the network). In this paper, we go below this $O(\log^2 n)$ memory, using approximation and parametrized complexity.

More specifically, our contributions are two-fold. We introduce a second parameter s, which is the space needed to encode a weight, and we design a silent polynomial-time self-stabilizing algorithm, with space $O(\log n \cdot s)$. In turn, this allows us to get an approximation algorithm for the problem, with a trade-off between the weight of the solution and the space used. For polynomial weights, this trade-off goes smoothly from memory $O(\log n)$ for an n-approximation, to memory $O(\log^2 n)$ for exact solutions, with for example memory $O(\log n \log\log n)$ for a 2-approximation.

Funding Support by ANR ESTATE

1 Introduction

1.1 Our questions

Context Self-stabilization is a technique to ensure fault-tolerance in distributed systems. It aims at designing systems that can recover from arbitrary faults. Silent self-stabilization consists in asking for the additional property that, once a correct configuration has been reached, the processors basically stop computing.

In the context of self-stabilization, the most studied measure of performance is the time to stabilize to a correct configuration. Another essential parameter is the space used by each processor. This parameter not only captures some notion of memory, but more remarkably, it captures the performance in terms of communication, as self-stabilizing algorithms communicate by taking a snapshot of the states of their neighbors (when they are described in the so-called *state model*, which is the most common model).

For silent self-stabilizing algorithms, this memory usage is tightly related to the space needed to locally certify that a configuration is correct. Such certifications, also called proofs, have been studied independently under the name of *proof-labeling schemes*. On the one hand, it is known that the space needed for the proof is a lower bound on the space required for silent stabilization. On the other hand, it is proved in [4] that one can always design an algorithm matching this lower bound (up to an additive logarithmic factor), even in the most asynchronous setting. Thus in some sense, one can always achieve optimal space. Unfortunately this general technique implies exponential stabilization time. An important question is then: what can we achieve in terms of space for self-stabilizing algorithms, with the constraint of polynomial time?

Approximation-memory trade-off We actually have two more precise question. The first question is: what can be done when we do not even have the space needed for a distributed proof? One way is to consider non-silent algorithms, that keeps exchanging information, after stabilization (see for example [3] that basically achieves $O(\log\log n)$ space for leader election on a ring, when the lower bound for silent stabilization is $\Omega(\log n)$). Another way is to consider relaxations of the tasks we consider. For example in this paper we are aiming at a trade-off between the memory used and the quality of the solution produced. More precisely, we want to design approximation algorithms for optimization problems, such that the larger the memory allowed, the better the approximation ratio.

Optimal space in polynomial time A second natural question is: when can we achieve optimal space in polynomial time. This is not possible in general. Consider for example the task of 3-coloring of 3-colorable graphs. The distributed proof uses only constant space, because the colors are enough for local checkability. But an algorithm in constant space cannot exist for this problem because one cannot break symmetry in constant space [1]. On the positive side, [2] shows that for various tree construction problems, one can match the space bound and have polynomial-time stabilization. In particular, one can get down to $\Theta(\log^2 n)$ for minimum spanning tree, which is optimal when the edge weights are in a polynomial range.

1.2 Our results

In this paper, we focus on the central problem of minimum spanning tree. Our main result is an approximation-memory trade-off for this problem. More precisely, we have a smooth

trade-off, between using memory $O(\log n)$ for an arbitrary spanning tree (that is a spanning tree whose weight is an n-approximation of the optimal weight), and memory $O(\log^2 n)$ for the minimum spanning tree (that is for the exact solution). A point of this trade-off is for example a 2-approximation (that is a spanning tree whose weight is at most twice the weight of the minimum spanning tree) in space $O(\log n \cdot \log\log n)$. The two extremes at $O(\log n)$ and $O(\log^2 n)$ are optimal, but we do not know if in the interval between these values our algorithm is tight, because there is no known lower bound for this regime.

▶ **Theorem 1.** *There exists a silent self-stabilizing approximation algorithm for minimum spanning tree, that stabilizes in polynomial time and has a trade-off between memory and approximation. This trade-off goes smoothly from exact solution in space $O(\log^2 n)$ to arbitrary solution in space $O(\log n)$.*

The main ingredient to achieve this result is an exact algorithm for MST (that is self-stabilizing, silent and polynomial-time), that uses $O(\log n \cdot s)$ space, where s is the number of bits used to encode an edge weight. This matches the performance of the $O(\log^2 n)$-space algorithm of [2], for polynomial weights, but applies to any weight range. We conjecture that this result is tight, because for both constant and polynomially large weights, an $\Omega(\log n \cdot s)$ bound is known [17, 18], and it seems likely that it is also the right answer for weights in the middle.

▶ **Theorem 2.** *There exists a silent self-stabilizing algorithm for minimum spanning tree, with $O(\log n \cdot s)$ memory, that stabilizes in polynomial time.*

1.3 Our techniques

Our first technique is a transformation of the weights, that allows to get a trade-off between the space used to encode a weight and approximation we can get using these transformed weights. The basic idea is the following: replace each weight by the position of its most significant bit. This way when we write weights in the memory, we use a space s that is $\log\log n$ bits instead of $\log n$ bits. Of course by this operation we loose some precision. Namely, as from the position we can recover the largest power of 2 that is smaller than the weight, we get a 2-approximation of each weight. As a consequence, if we feed these "new weights" to an exact algorithm for minimum spanning tree, we will get a 2-approximation. We design an extension of this technique, that allows to get the full trade-off between space and approximation, while still considering only simple computations on the binary representation of the weights.

The rest of the paper consists in designing an exact self-stabilizing algorithm for minimum spanning tree in space $O(\log n \cdot s)$. Note that, if we plug the $s = \log\log n$ associated to a 2-approximation in the previous paragraph, we get space $O(\log n \log\log n)$, which is what we claimed in the previous subsection.

This exact algorithm does not follow the usual design of silent self-stabilizing algorithms. A silent algorithm typically stores some key pieces of information, *while* it is building the output, *e.g.* selecting edges of a tree. These pieces of information form a certificate of correctness that allows, during and after stabilization, to check whether the construction is correct. And if the construction is correct then the output is correct. (This is for example the way the $O(\log^2 n)$-space algorithm of [2] is designed, book-keeping the important information of a Borůvka-inspired algorithm.) Unfortunately, it appears that this approach is difficult (if not impossible) to use, when one wants to go below the $\log^2 n$ space for minimum spanning

tree.[1] What we do instead, is to first build a minimum spanning tree, and, once it is finished, to certify it. This paper is, as far as we know, the first occurrence of such a modular approach. The certification we use comes from [17], where designing a self-stabilizing algorithm that actually builds this certification was left as an open problem.

1.4 Outline

We start in Section 2, with a description of the model. In Section 3, we describe the general structure of our algorithm. Then in Section 4, 5, and 7, we describe the different components of our algorithm. Section 6 is a high-level description of the certification of [17] that we use in Section 7. In the appendix, we justify our two-phases approach by discussing why building and certifying at the same time is difficult.

2 Model

We consider that a network is represented by an undirected connected graph $G = (V, E)$ where V is a set of processors (or nodes), and E is a set of edges that represent communication channels. We denote the number of nodes by n. We consider that every node v is given a unique identifier ID_v. In this paper, the edges are weighted, and we want to compute an approximation of a minimum spanning tree. A k-approximation of a minimum spanning tree is a spanning tree whose weight is not larger than k times the weight of the minimum spanning tree. Note that in our definition of approximation, we do not relax the requirement of acyclicity, thus it is a kind of approximation that is different from the one of the spanners litterature.

State model Our algorithm is designed for the classic *state model* [10]. In this model, every node has a state, and communication between neighbors is modeled by direct reading of states instead of an exchange of messages. This state is the mutable memory, that is the part of the memory that can be modified, and also the one that is counted when we consider space complexity. There is also the non-mutable memory, that contains for each node, its identity, the weights of the adjacent edges, as well as the code of the algorithm. The tuple of all the states of the network is called the *configuration*, and the execution of an algorithm is therefore described by a sequence of configurations.

In the state model, an algorithm is usually described as a set of rules. A rule basically states that if the local view of a node satisfies some property, then the node can change its state to a specified new state. Here by "local view" we mean the state of a node, the states of its neighbors, the ID of the node, and the weights of the adjacent edges. If the property of the node's view is satisfied, then we say that the node is *enabled*. The asynchronism of the system is modeled by a scheduler that chooses, at each step, a non-empty subset of enabled nodes, that are allowed to apply a rule. We consider the harshest scheduler, the *distributed unfair scheduler*, which has no further constraint for the choices it makes (distribution and fairness restricts choices of the scheduler respectively over space and time). There exists a variety of other schedulers, each of them being weaker than ours, see [12].

To compute time complexities, we use the definition of *round* of [11]. This definition captures the execution rate of the slowest process in any execution. The first round of an execution ϵ, noted ϵ', is the minimal prefix of ϵ such that every node that was enabled in the

[1] We justify this claim in an appendix to the paper.

initial configuration, either has taken a step or is not enabled anymore. Let ϵ'' be the suffix of ϵ such that $\epsilon = \epsilon'\epsilon''$. The second round of ϵ is the first round of ϵ'', and so on.

Distributed proof, proof-labeling schemes and silent stabilization Given a property, for example 'a set of pointers defines a spanning tree', a *distributed proof* (or *distributed certificate*) is a labeling of the nodes that certifies that the property is satisfied. It is usually presented as a *proof-labeling scheme* [18]. In such a scheme, the first element is an oracle, called the *prover*, which provides each node with a label. The second element of a scheme is a *verification algorithm*. This algorithm, run at a node v, takes the view of v and decides whether to *accept* of to *reject*. A scheme is correct for a property P, if (1) for any configuration satisfying P, there is a way for the prover to make all nodes accept, and (2) for any configuration that is not satisfying the property, there is no labeling such that all the nodes are accepting. The performance of a scheme is measured by the size of the labels in number of bits (all labels have the same size). The notion of distributed proof is tightly related to the concept of *silent self-stabilizing algorithm*. An algorithm is *self-stabilizing* if after some finite time, called the *stabilization time*, it is in a correct configuration for the problem, and stays in correct configurations afterwards. Such an algorithm is *silent*, if the algorithm reaches a correct configuration and then stays silent: it does not change the states anymore, or in other words, no node is enabled. To be sure to be in a correct configuration, a silent self-stabilizing algorithm has to keep some certification of the correctness. This certification ensures that, if the configuration is not correct, then at least one node will detect it, be enabled, and start the recovery (for example launching a reset). This is basically the same as the notion of distributed proof above [4], except that the proof is not given by an oracle: it is built by the algorithm.

3 General description

Our algorithm is made of several components. Basically, we have several algorithms that will operate one after the other, in order to reach a configuration with a minimum spanning tree certified by a distributed proof. The algorithms are designed to work if they start from a clean situation (for example our first algorithm expects its variables to be empty) and we have a reset mechanism that will go back to such a situation if one of our algorithms detects a problem.

General design and requirements on the components The strategy of aiming for a distributed proof simplifies the design of the algorithm. Indeed, if something goes wrong in the computation because of the initial configuration, then we have to face two rather simple situations. In the first case, one of our algorithms detects the error, for example because there is an obvious inconsistency between the neighbors' states. In the second case, the problem is subtle enough to not be detected during the run of our algorithms, but then if the output is not correct, the distributed proof that is built cannot be correct either. In both cases the error is detected, and a reset is launched. In other words, either there is something obviously wrong that is caught on the fly, or there is something that is more subtle, and it is caught at the end.

The difficulties that remain are the same as for any self-stabilizing algorithm. First one has to ensure, that, for any schedule of the steps, a situation that comes from a clean configuration will not end up being detected as an inconsistency. If this were to happen, then the scheduler could make the algorithm go through a reset infinitely often, and we would not

stabilize. Second, we have to make sure that the algorithm does not get stuck in a position where no node can be activated.

The components We have three main components: one algorithm that builds the minimum spanning tree (detailed in Section 5), one algorithm that takes this tree and certifies it (detailed in Section 7), and a reset algorithm that can erase everything and go back to a clean configuration. For the reset, we take a solution from the literature. Devismes and Johnen [9] recently propose a cooperative (that is tolerating multiple simultaneous initiators) silent self-stabilizing reset algorithm that satisfy our constraints in terms of daemon (distributed and unfair), stabilization and completion time (both polynomial in rounds) and spatial complexity ($O(\log n)$ bits).

Also, for the approximation algorithm of Theorem 1, we have to transform of the weights. This transformation consists in replacing each weight by a smaller weight. But, as we have limited space, we cannot store all the transformed weights of the adjacent edges in the state (there can be as many as order of n edges adjacent to a node). Instead, every time a step is taken, the node recomputes the new weights to know which rule applies. (Remember that the only space we take into account is the space used to write the new state, not the space used for the computation of this new state. As said in the introduction, the focus is more on the communication than on the actual space needed for the computation of each processor.)

Finally, as our algorithm has several components, each of them using basic blocks from the literature, we chose to not describe the algorithm in the most formal way with explicit sets of rules. We describe the algorithm on a high-level, but highlighting the key aspects and difficulties.

4 Approximation-memory trade-off

In this section, we show how we can replace the input weights with approximations to decrease the number of bits used to encode them, while preserving guarantees on the MST computed. We prove the following result:

▶ **Lemma 3.** *There exists a simple parametrized transformation of the weights that allows for a trade-off between space needed to encode weights, and approximation. In particular, one can get:*

- *Exact solution, with weight encoded on* $\log n$ *bits.*
- *2-approximation, with weights encoded on* $\log \log n$ *bits.*
- *No weight guarantee, with weights encoded with a constant number of bits.*

Proof. The core idea of weight transformation is to group the weights into buckets of similar size, that will be assigned the same transformed weight. The larger the group, the less bits needed to encode the group name, but the larger the rounding error on each weight. The basic idea is to use exponential bucketing. For example, with exponent 2, a weight x, will be a bucket p, if $2^{p-1} < x \leq 2^p$. This way, every weight is at most doubled, and there are $\log n$ different weights instead of n, which means it can be encoded on $\log \log n$ bits. To adjust the trade-off between approximation and space, we could change the exponent: replacing 2 by a larger number, means worse approximation but better size, and smaller number means better approximation, but larger size. But this has two drawbacks. First, this cannot be used to get a trade-off that goes all the way to the exact solution. Indeed even taking the exponent very close to 1, one does not get all buckets of size 1. Second, this computations imply a lot of rounding, and even if we do not restrict the computational power of the nodes, it is

more elegant to have a transformation that works with simple manipulations of the binary representation of the weights. For example, the bucketing with a power of 2 is efficient, as one can decide the bucket by only considering the position of the most significant bit. We then use a slightly different technique.

For this technique, we consider *milestones* which are simply numbers between 1 and n, such that in the transformation, each weight is rounded up to the closest milestone. For simplicity, we suppose that n is a power of 2. As we are dealing with approximation, and interested in asymptotics, this is harmless. In the following claim, *approximation* x, means that the rounding to the next milestone implies that the weight is multiplied by at most x.

▷ Claim 4. For every integer k in the range $[-\log\log n, \log n]$, there exits a set of milestones with the following properties:

- for $k \geq 0$, approximation $1 + \frac{1}{2^k}$ and size $2^k(\log n - k + 1)$.
- for $k \leq 0$, approximation 2^{2^k} and size $\frac{\log n}{2^k} + 1$.

We start from the set of milestones described above, that is $1, 2, 4, 8, 16, ..., 2^{\log n - 1}, 2^{\log n}$, which corresponds to $k = 0$. For $k > 0$, we will add milestones, and for $k < 0$ we will remove milestones.

Let us start with $k > 0$. The construction is iterative. From k to $k + 1$, we take every interval between two milestones that is non empty, and take the number in the middle. For example, for $k = 1$, the set of milestones is: $1, 2, 3, 4, 6, 8, 12, 16, ..., 2^{\log n - 1}, (3/2)2^{\log n - 1}, 2^{\log n}$. Note that the construction is well-defined as the number of elements in a non-empty interval is odd in the $k = 0$ construction, and this property is preserved by the construction. At each iteration, the number of milestones is at most doubled, but as some intervals are empty, the precise increase is smaller. It is not difficult, to compute that the precise number of milestones is: $2^k(\log n - k + 1)$. The approximation ratio is bounded by the largest ratio between two consecutive milestones, which is $1 + \frac{1}{2^k}$ (it is reached for a milestone of the original $k = 0$ construction and its successor).

We now tackle the case $k < 0$. Again we use an iterative construction. From k to $k - 1$, we remove one in two milestones, except for the last one. For example, $k = -1$ corresponds to $1, 4, 16, ..., 2^u, 2^{\log n}$, where u is the largest even integer strictly smaller than $\log n$. At step k there are at most $\frac{\log n}{2^k} + 1$ milestones, and the approximation is 2^{2^k}. Which finishes the proof of the claim.

Note that using these milestones, we get the results claimed in the lemma. For $k = 0$, we have the 2-approximation in space $\log\log n$. For $k = \log n$, we get the exact solution, as the set of milestones contains all the integers between 1 and n, and we use $\log n$ space. For $k = -\log\log n$, we have an n-approximation, but use just a constant number of bits.

Also, the computations only deal with sums of powers of 2, thus they can all be done easily on the binary representations of the weights. ◀

5 MST construction algorithm

In this section, we give a distributed algorithm for minimum spanning tree. This algorithm is not self-stabilizing, and we will build a second algorithm in Section 7, to add a labeling to the nodes, that allows silent stabilization. As our main goal is to use small space, we do not optimize the time of this construction algorithm, and keep it as simple as possible.

▶ **Lemma 5.** *There exists a distributed algorithm in space $O(s + \log n)$ that builds a minimum spanning tree in polynomial time.*

Proof. Our algorithm is a distributed version of Kruskal's algorithm. Remember that Kruskal's algorithm sorts the edges by increasing weight, and then adds the edges to the tree one by one, discarding any edge that would close a cycle. There are several modifications to perform in order to make this algorithm work in our framework.

First, to consider the edges by increasing weights, we work in phases, each phase corresponding a specific value that a weight can take. We have a phase for the possible edges of weight 1, then a phase for those of weight 2, etc.

Then in order to have these phases somehow synchronized on the whole network, and to avoid simultaneous additions of edges of the same weight (which could create cycles), we use a token. This token visits the nodes of the graph, and only the node with the token will be allowed to add edges to the graph. The token transport the information about the weight of the current phase. To make the token visit all the nodes we need to build a spanning tree beforehand, and we make the token traverse the tree. We use the same spanning tree construction and token circulation as in [5], inspired by the tree algorithm of [8] and the token algorithm of [19]. Indeed, these algorithms ensure to stabilize to a token circulation on a spanning tree in $O(n)$ rounds using only $O(\log n)$ bits under our settings.

Finally, a node must be able to decide locally whether adding a specific edge would close a cycle or not. To do so we will keep for a every node a *name*, which is the smallest identifier of the nodes it is connected to, by the current set of selected edges. This way, a node can safely add an edge, if its name is different from the name of the other node. To maintain this name, we will need to perform a traversal of a part of the tree every time we add an edge. This is also known to be doable in our setting [6].

The space complexity is in $O(s + \log n)$, because we just need to store a constant number of IDs, weights and additional $O(\log n)$ objects. The time complexity in terms of rounds is polynomial, because there are n phases, n nodes to visit with the token, and the traversal of a fragment takes at most $O(n)$ rounds. ◀

Augmented MST Actually, for the next steps (described in Section 7), we will need a bit more than the minimum spanning tree. We want to have for each node: an orientation to an arbitrary root, the number of nodes that are descendants of this node (including the node itself), and the total number of nodes in the graph. These are easy to compute by simple traversals.

6 Distributed proof of MST

The second part of our algorithm, that makes it self-stabilizing, consists in labeling the nodes with a distributed proof. This labeling certifies the correctness of the minimum spanning tree. It comes from a proof-labeling scheme described in [17], and we just sketch it here. For a intuitive description of the main ideas behind the scheme, we refer to [13].

▶ **Lemma 6** ([17]). *There exists a distributed proof of size $O(\log n \cdot s)$ for minimum spanning trees.*

Sketch of the scheme and of the proof. The labeling is actually in two parts. The first part certifies the acyclicity of the tree. It is well known that acyclicity can certified with $O(\log n)$ bits [18], for example with each node storing its number of descendants, thus we focus on the second part.

The (second part of the) scheme has a recursive shape. Let us first describe the top-most level of it. The prover chooses a node to be the center of the tree, and orients the tree towards

this node such that it becomes a root. This node has some x subtrees, that is, if we remove this center from the graph, there would be x trees. The prover gives a distinct number to each subtree from 1 to x. Every node (except the center) is labeled with the number of its subtree. Also every node is given the maximum weight on its path to the center along the tree. It is rather easy to check that the correctness of these pieces of information can be checked locally.

To show why these pieces of information are useful, consider a node u of a subtree A, that is adjacent in the graph (but not in the tree) to a node v of a subtree B. Thanks to the subtree numbers in their labels, the nodes u and v know that they do not belong to the same subtree. We claim that they can also check whether adding (u, v) to the tree could result in a smaller weight tree (which would contradict the fact that the selected edges form an MST). This is based on the following simple equivalence. Adding (u, v) leads to a lighter tree, if and only if, the path from u to v (in the tree) has an edge that is heavier than (u, v). This path must go through the root, because the nodes are in different subtrees. Thus the maximum weight on this path is either the maximum weight from u to the root, or from v to the root, and the nodes have access to this information: these weights are in their labels.

To allow the nodes to test for all the non-selected edges, we need to handle adjacent nodes that *are* in the same subtree. We do so by choosing recursively a center in each subtree, and providing the same information as for the top level. That is every node will have information regarding the first center, its second center (that is the center of its subtree of the first center), its third center (that is the center of its subtree of the second center) etc. until the level where it is itself a center. Thanks to this recursive structure, for every pair of nodes, there is a level of the recursion for which they are assigned to two distinct subtrees (or to a subtree and its root), and the checking can be done in the same way as for the top-most level.

To be sure to use only $O(\log n \cdot s)$ space, for all this information about all the centers of a node, we need the above scheme to have two properties. First, we need the centers to be placed in a balanced manner. Precisely, we want the center of a (sub)tree T to be at a node such that every subtree has size at most $|T|/2$. (Such a node always exists and can be computed in a simple way, as described in the next section.) This balance for the centers implies that there is at most $O(\log n)$ centers per node. Second, we need that for each node, the concatenation of all its subtree numbers is not too long. To get this, it has been proved (see [15]) that it is enough that, for each center, the subtrees are numbered by decreasing number of nodes (the largest subtree gets number 1, the second largest gets number 2, etc.).

Also, remember that we need to have an orientation towards the center, for each of the $O(\log n)$ centers a node has. This takes $O(\log^2 n)$ bits if we use the ID of the parent as a pointer. Instead, we have only one orientation of the full tree encoded by IDs, and the other orientations are encoded with respect to this original orientation. Concretely, every node will store whether the center of the current level is in the direction of its parent in the original orientation, or if it is in the direction of one of its children. Surprizingly, this is enough to describe and check each orientation, and it takes only constant number of bits per level. (See [17] or [13], to see for example why not knowing which child is the parent is not a problem.) ◀

7 Certification labeling algorithm

In this section, we describe an algorithm that builds the labeling of Section 6. Remember that thanks to Section 5, we start with a tree that has an orientation toward a root and

whose nodes are labeled with the number of nodes in their subtrees. Thus the first part of the labeling of Section 5 is already present.

▶ **Lemma 7.** *Given a tree with orientation to a root and subtree sizes, there exists an algorithm that builds the labeling of Section 6 in space $O(\log n \cdot s)$ and polynomial time.*

Proof. We first describe the algorithm, and then highlight some key properties. The algorithm takes as input a tree, and certifies it as a minimum spanning tree. The construction follows the recursive description of the labeling of Section 6.

Before any computation, we do a copy of the pointers, subtree number, etc. We will modify the copy, but we need to keep the original labels. Let us first describe the computation of a center. (This description is for the first center, we explain later how to adapt it to the other phases of the labeling.) Basically, we will move the root of the tree until it satisfies the property of a proper center. (Remember that this property is that the center separates the tree into parts that have size at most half the size of the full tree.) Our first candidate for a center is then the root of the tree. Thanks to the subtree sizes that every node holds, the root can detect whether it is in a central position or not. Indeed, as the subtree size of the root is the number of nodes in the full tree, the root can easily check whether its neighbors have subtree sizes at most half this number. If the root is not in a central position, then we transfer the root to the neighbor having the largest subtree size. This transfer of the root implies several computations:

- the old root designates the new root
- the old root orients its pointer to the new root
- the new root, erase its pointer and takes the root label
- the old root takes as new subtree size: its old subtree size, minus the old subtree size of the new root
- the new root takes as subtree size the old subtree size of the old root.

Thanks to this step, we are in the same position as before (that is, we have a tree with a correct orientation, and correct subtree sizes), but in addition, the root is in a more central position, in the sense that the largest subtree size is smaller than before the transfer. After $O(n)$ such moves, the root is in a central position.

Once the center is validated, we have to label each node with: the orientation to the center, the maximum weight on the path to the center, and the subtree number. There is a difficulty here. Remember that only the center can decide which subtree gets which number, because these numbers depend on the relative sizes of all the subtrees. Thus the center has to announce the subtree numbers *e.g.* "the subtree with root ID_v has number 3". It is not possible to announce all these numbers at once. Indeed, the list of these numbers can have length of order n. Instead the center will announce a first pair identifier-number, then wait for the node with this identifier to confirm this information, and then go to the second etc. Once the root of a subtree receives its subtree number, it broadcasts this information to all its subtree using a snap-stabilizing (*i.e.,* self-stabilizing with a stabilization time of 0 rounds) Propagation of Information with Feedback algorithm. Bui *et al.* [6] provide such an algorithm with constant space requirement and polynomial completion time (in rounds) on trees, that meets our requirements. In the same wave, the maximum weight to the center is computed by keeping and updating the maximum weight seen so far, while descending in the tree. The orientation is even easier to store: just copy the current orientation.

Once the broadcast waves have come back to the root of the subtree, every node has; in its label, all the information it needs to store for this phase. And then we can start

immediately a new recursive call, that is looking for the next center. Indeed we have a tree, with a root, with a proper orientation, and with correct subtree numbers.

Let us now highlight some key points of the algorithm.

- Once a center has finished launching the computations in each subtree, it becomes silent, and will not change its state, except if there is a reset.
- As soon as two adjacent nodes (in the graph) become centers, they can start checking their labelings to see if the distributed proof makes sense, at least for this edge (and launch a reset if it is not the case).
- Once the root has launched the computation in the subtrees, these subtrees will run independent computations (except for the final checking mentioned in previous item). This means that the scheduler can delay the computation in one subtree, by making a series of recursive calls in the other subtrees, but at some point these recursive calls will end up by having all nodes as centers, thus disabled, and the scheduler will have to enable the remaining nodes.
- Suppose that we are in a subtree, looking for the center for the second recursive call. The main center is already chosen, and thanks to the pieces of information stored, the nodes can check that it exists. But they cannot check whether it was placed in a central position, because we are reusing the subtree size variables. Thus if we start from an initial configuration, that corresponds to this situation, we are not following the correct construction described in Section 6. This means that we could end up with a larger memory than what we claimed. What we do is that we control the size used. If we end up using more than $\alpha \cdot \log n \cdot s$ bits, for a constant α large enough to allow correct computations, then we launch a reset. Note that it may actually be the case that the centers are not perfectly placed, but that their positions are good enough to ensure that the memory size does not cross the $\alpha \cdot \log n \cdot s$ limit; in this case we do not detect it, and the outcome is correct.

◀

8 Obstacles to a simultaneous proof labeling

As argued in the first sections of the paper, our algorithm has an uncommon shape: first it builds a minimum spanning tree, and then it certifies it. In this section, we discuss a few obstacles to design an algorithm that would simultaneously build and certify a minimum spanning tree with memory $O(\log n \cdot s)$. More precisely, we show why natural modifications of the algorithm based on Borůvka algorithm do not work.

Quick description of the Borůvka-based scheme. Let us remind Borůvka's algorithm to build a minimum spanning tree. Every node begins as a so-called *fragment*. Then there are phases. At each phase, every fragment finds the lightest edge that has exactly one endpoint in the fragment and one endpoint outside the fragment. This edge is added to the tree, except if two fragments choose to link one to the other by two different edges of same weight, in which case, only one edge is taken. Then the two fragments merge to form one fragment for the next phase. After $O(\log n)$ phases, there is only one fragment left and it is a minimum spanning tree of the graph. This type of algorithm is adapted to a distributed setting, as it does not need a lot of coordination between the nodes and fragments.

The distributed proof that corresponds to this algorithm, and that can built at the same time as the tree itself is the following. Every node is given a table with one cell for each phase of the algorithm. In each cell, there is the name of the edge taken by the fragment

of the node for this phase, as well as the weight of this edge, and the local encoding of a spanning tree pointing to this edge. (The edge is given by the identifiers of its endpoints.) The nodes can check locally that the designated edge exists, and that indeed it is the lightest out-going edge. Also they check that the different cells are consistent.

Characteristics. Let us now list some characteristics of this scheme.

1. The algorithms proceeds in phases where fragments are merged.
2. The certification is monotone: at each phase, one concatenate information to the label.
3. The fragments are identified using the identifiers of the nodes.
4. There are $\Theta(\log n)$ phases in the worst case.
5. The number of fragment that are merging at each phase is not controlled. That is, there can be between 2 and $\Theta(n)$ fragments merging in one phase.
6. The places where the fragments merge is not controlled. More precisely, the merge edges of a same fragment can be far apart.

Discussion. We will only discuss algorithms that follow Item 1, that is, algorithms based on fragments fusions. Of course this narrows the discussion, but note that: (1) most distributed MST algorithms are of this form, and (2) the $\Omega(\log n \cdot s)$ lower bounds, suggests that the structure of approximately $\log n$ levels is somehow compulsory. Also, we restrict to schemes that are monotone (Item 2), as it seems to be a natural requirement when looking for algorithms that build and certify at the same time.

Now having both Item 3 and Item 4, immediately leads to a $\Theta(\log^2 n)$ factor, that we want to avoid. As it seems difficult to avoid having a logarithmic number of levels, in the remaining we avoid using identifiers.

The first obstacle is the following. It seems necessary to give names to the fragment, such that the nodes can check connectivity, at the different steps. As we do not use identifiers, we have to certify that different fragments have different names. For example take a phase in which several fragments merge ; these fragments must have different names. We have the following claim.

▷ Claim 8. In this case, if Item 5 and Item 6 hold, then we may need proof of size $\Theta(n)$.

Consider the following scenario. There is a fragment F, that is a long path, with two endpoints, left and right. There are k fragments merging with F on the left, and k other fragments merging with F on the right. Also the fragment names are numbers between 1 and $2k$, distributed between right and left in an arbitrary manner. Then to check whether the names are all different, we need to decide whether the set of names on the right and on the left are disjoint. This boils down to the disjointness problem in communication complexity, and we know that we cannot do it without using space $\Theta(k)$. (More formal reductions between distributed proof and communication complexity appear in [7, 14, 16].) As k can be as large as $\Theta(n)$, we get the claim. In a nutshell: if the scheme does not use IDs, control neither the number of fragments merging, nor the merge edges, then ensuring that the names are different requires large proofs.

A way to avoid this obstacle would be to avoid Item 5, and to control the number of merged fragments. For example we could limit a merge to two fragments. But then we would face a second obstacle.

▷ Claim 9. If only two fragments can merge, then there can be $\Omega(n)$ phases.

Consider a star. At every merge, at most one new node can merge with another fragment, as only the center of the star is in the position of merging. For a star with n nodes, we would need $\Theta(n)$ phases. That is, with this strategy, we loose Item 4. Then, even if we can go down to only one bit per phase, we still have a total length of $\Theta(n)$ for the concatenation of the fragment numbers.

A second way to avoid the first obstacle is to control the names for the fragment. Indeed the reduction to communication complexity holds only if the names can be arbitrary (between 1 and $2k$). If we could ensure that the name of the fragments on the left side begin with 0, and the ones in the right side begin with 1, then we could check locally without transferring information from one side to the other. But in this case, we face a third problem: it is difficult to get the concatenation of all the names to be in $O(\log n)$ instead of the basic $\Theta(\log^2 n)$, if we put such strong constraints on the names used. For example the reverse size numbering we used does not work here, as there is no reason why one side would get only the smaller fragments and the other would get only the larger ones.

9 Conclusion

This paper presents the first polynomial silent self-stabilizing algorithm for MST construction using $O(\log n \cdot s)$ bits of memory (where n is the size of the network and s is the space to encode a weight). The motivation behind the use of two parameters, is to go below the existing solutions using $O(\log^2 n)$ space.

Using as another tool a bucketing approach that reduces the space needed to encode weights, this algorithm allows us to provide a tiny memory approximation algorithm for the MST construction. We obtain a trade-off that goes smoothly from memory $O(\log n)$ for an n-approximation, to memory $O(\log^2 n)$ for exact solutions, with an interesting milestone of $O(\log n \log\log n)$ bits for a 2-approximation.

We adopt a somewhat unusual approach in the silent self-stabilization area since our algorithm operates in two phases. It first builds a (candidate) MST and then certifies it using a proof (that is not related to the algorithm) rather than building the proof along the execution of the algorithm (by keeping some trace of this execution). We believe that this modular approach is the key for breaking the $O(\log^2 n)$ barrier, as discussed in Section 8.

References

1 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.

2 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.

3 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.

4 Lélia Blin, Pierre Fraigniaud, and Boaz Patt-Shamir. On proof-labeling schemes versus silent self-stabilizing algorithms. In *Stabilization, Safety, and Security of Distributed Systems - 16th International Symposium, SSS 2014*, pages 18–32, 2014. doi:10.1007/978-3-319-11764-5_2.

5 Lélia Blin, Fadwa Boubekeur, and Swan Dubois. A self-stabilizing memory efficient algorithm for the minimum diameter spanning tree under an omnipotent daemon. *J. Parallel Distrib. Comput.*, 117:50–62, 2018. doi:10.1016/j.jpdc.2018.02.007.

6 Alain Bui, Ajoy Kumar Datta, Franck Petit, and Vincent Villain. Snap-stabilization and PIF in tree networks. *Distributed Computing*, 20(1):3–19, 2007. doi:10.1007/s00446-007-0030-4.

7 Keren Censor-Hillel, Ami Paz, and Mor Perry. Approximate proof-labeling schemes. In *Structural Information and Communication Complexity - 24th International Colloquium, SIROCCO 2017*, pages 71–89, 2017. doi:10.1007/978-3-319-72050-0_5.

8 Ajoy Kumar Datta, Lawrence L. Larmore, and Priyanka Vemula. An o(n)-time self-stabilizing leader election algorithm. *J. Parallel Distrib. Comput.*, 71(11):1532–1544, 2011. doi:10.1016/j.jpdc.2011.05.008.

9 Stéphane Devismes and Colette Johnen. Self-stabilizing distributed cooperative reset. *CoRR*, abs/1901.03587, 2019. arXiv: 1901.03587.

10 Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000. ISBN 0-262-04178-2.

11 Shlomi Dolev, Amos Israeli, and Shlomo Moran. Resource bounds for self-stabilizing message-driven protocols. *SIAM J. Comput.*, 26(1):273–290, 1997. doi:10.1137/S0097539792235074.

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

13 Laurent Feuilloley. Note on distributed certification of minimum spanning trees. arXiv: 1909.07251, 2019.

14 Laurent Feuilloley, Pierre Fraigniaud, Juho Hirvonen, Ami Paz, and Mor Perry. Redundancy in distributed proofs. In *32nd International Symposium on Distributed Computing, DISC 2018*, pages 24:1–24:18, 2018. doi:10.4230/LIPIcs.DISC.2018.24.

15 Cyril Gavoille, Michal Katz, Nir A. Katz, Christophe Paul, and David Peleg. Approximate distance labeling schemes. In *Algorithms - ESA 2001, 9th Annual European Symposium*, pages 476–487, 2001. doi:10.1007/3-540-44676-1_40.

16 Mika Göös and Jukka Suomela. Locally checkable proofs in distributed computing. *Theory of Computing*, 12(1):1–33, 2016. doi:10.4086/toc.2016.v012a019.

17 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.

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

19 Franck Petit and Vincent Villain. Time and space optimality of distributed depth-first token circulation algorithms. In *Distributed Data & Structures 2, Records of the 2nd International Meeting (WDAS 1999)*, pages 91–106, 1999.