

Silent MST Approximation for Tiny Memory

Lélia Blin, Swan Dubois, Laurent Feuilloley

▶ **To cite this version:**

Lélia Blin, Swan Dubois, Laurent Feuilloley. Silent MST Approximation for Tiny Memory. SSS 2020 : The 22th International Symposium on Stabilization, Safety, and Security of Distributed Systems, Nov 2020, Austin, TX / Virtual, United States. pp.118-132, 10.1007/978-3-030-64348-5_10 . hal-03140584

HAL Id: hal-03140584
https://inria.hal.science/hal-03140584

Submitted on 13 Feb 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Silent MST approximation for tiny memory*

Lélia Blin,[1][0000−0003−0342−9243], Swan Dubois[1], and Laurent Feuilloley[2][0000−0002−3994−0898]

[1] LIP6, Sorbonnes University
[2] DII, Universidad de Chile

Abstract. In this paper we show that approximation can help reduce the space used for self-stabilization. In the classic *state model*, where the nodes of a network communicate by reading the states of their neighbors, an important measure of efficiency is the space: the number of bits used at each node to encode the state. In this model, a classic requirement is that the algorithm has to be *silent*, that is, after stabilization the states should not change anymore. We design a silent self-stabilizing algorithm for the problem of minimum spanning tree, that has a trade-off between the quality of the solution and the space needed to compute it.

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 (and is actually also called *the memory*), but more remarkably, it captures the performance in terms of communication, as self-stabilizing algorithms communicate by reading 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. Indeed, after stabilization, a silent algorithm is only checking that the configuration is correct via reading its neighbors' states, which is exactly what a distributed proof is made for. On the other hand, it is proved in [5] that one can always design an algorithm matching this lower bound (up to an additive logarithmic

* Support by ANR ESTATE

factor), even in the most asynchronous setting. Thus in some sense, one can always achieve optimal space.

There are two issues to this situation. First the general technique of [5] is inherently exponential in time: it basically consists in looking for the distributed proof via an exhaustive search. Second, the space required for silent stabilization can be simply be too large for applications.

Approximation-memory trade-off The core of our paper is to give a solution for the second problem, which can be rephrased as: what can be done when we do not even have the space needed for a distributed proof? One technique is to consider non-silent algorithms, that keep changing their states. For example, [4] achieves $O(\log \log n)$ space for leader election on a ring, when the lower bound for silent stabilization is $\Omega(\log n)$ (where n is the number of nodes in the network). In this paper, we make the choice of keeping the silence property, but to be less demanding on the quality of the solution. More precisely we are aiming at a trade-off between the memory used and the quality of the solution produced, that is, we want to design approximation algorithms for optimization problems, such that the larger the memory allowed, the better the approximation ratio. To our knowledge this is the first time approximation is used to reduce memory usage for self-stabilization (although it has recently been proved fruitful in the more restricted context of proof-labeling schemes [10, 16]).

Optimal space in polynomial time Now a second question, which follows from the exponential-time algorithm of [5], is: when we can afford the optimal space to compute an exact solution, can we get it in polynomial time? The answer is no in general. Consider for example the task of 3-coloring a 3-colorable graph. The distributed proof uses only constant space, because the colors are enough for local checkability. On the other hand, it is known that no algorithm can compute a 3-coloring in constant space. Indeed, in order to perform even a minimal symmetry breaking (such as having two nodes with two different outputs), an algorithm needs strictly more than constant space [2] (actually $\Omega(\log \log n)$ bits are necessary [8]). On the positive side, [3] shows that for various tree construction problems, one can match the optimal space bound and have polynomial-time stabilization. In particular, one can get down to $\Theta(\log^2 n)$ bits for minimum spanning tree, which is optimal when the edge weights are in a polynomial range. As we will see, we can improve on this, as a side result of our approximation algorithm.

1.2 Our results

In this paper, we focus on the central problem of minimum spanning tree (MST). Our main result is an approximation-memory trade-off for this problem. The

theorem below, and all the results of this paper hold under the classic assumption that the edge weights are in $[1, n]$, where n is the size of the network.[3]

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 from space $O(\log^2 n)$ for a minimum spanning tree to space $O(\log n)$ for a simple spanning tree.*

The precise trade-off has a complicated expression, thus we do not write explicitly here. It is given in Lemma 1. The two extreme values, $O(\log^2 n)$ for an MST and $O(\log n)$ for a simple spanning tree are optimal (see [22, 23] for the lower bounds). We get a smooth trade-off between these extremes, with for example $O(\log n \log\log n)$ space for a 2-approximation.

One of the two ingredients to achieve this result is an exact algorithm for MST, which is self-stabilizing, silent and polynomial-time, and uses $O(\log n \cdot s)$ space,[4] where s is the number of bits used to encode an edge weight.[5]

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

It is known that an MST requires $\Omega(\log n \cdot s)$ space [22]. Therefore our algorithm improves on the state-of-the-art by proving that, even with a parametrization by s, MST is part of the set of problems that can be solved in optimal space *and* polynomial-time.

1.3 Our techniques

Our algorithm has two main ingredients: the exact algorithm that we have already mentioned and a technique to transform the weights. The weight transformation changes the original weights into approximated weights that can be encoded in smaller space. Then we basically feed these approximated weights to the exact algorithm and get as a result an approximate solution. The better the approximation of the weight, the better the approximation of the final solution, but the larger the space used.

The weight transformation (Lemma 1) takes as input a weight in $[1, poly(n)]$, encoded on $\Theta(\log n)$ bits, and outputs a weight in a smaller range, hence using less bits of memory. The simplest form of the technique 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 exponentially less space: s will be in $O(\log\log n)$ instead of $\Theta(\log n)$. Of course by this operation we loose some precision. Namely, we only have the information to recover a 2-approximation of each weight. Now

[3] Assuming the maximum to be n and not poly(n) allows to have cleaner proofs without additional constants, but the asymptotic results are also correct for polynomial weights.

[4] Here and everywhere in the paper, $\log n \cdot s$ should be read as $(\log n) \times s$.

[5] Note that as we assume the weights are polynomial in n, s is in $O(\log n)$.

if we feed these "new weights" to an exact algorithm for minimum spanning tree, then we will get a 2-approximation, and using much less space. We design an extension of this technique, that allows to get the whole trade-off between space and approximation. This extension is more complicated, but is still based on manipulations of the binary representation of the weights.

The rest of the paper consists in designing the exact self-stabilizing algorithm for minimum spanning tree in space $O(\log n \cdot s)$. 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.* while selecting the edges of the MST. 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 [3] is designed, book-keeping the important information of a Boruvka-inspired algorithm. Unfortunately, it seems that this approach is difficult if not impossible to use when one wants to go below $\log^2 n$ space for minimum spanning tree. Instead, we use a two-phase approach: we first build a minimum spanning tree, and, once it is finished, we certify it. This paper is, as far as we know, the first occurrence of such a modular approach. The certification we use is the proof-labeling scheme of [22]. As a side result we answer a question of [22] where designing a self-stabilizing algorithm using 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 [22] that we use in Section 7.

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. Every node v is given a unique identifier ID_v, and every edge has a weight. Both identifiers and weights are polynomial, that is, they are integers in $[1, poly(n)]$. The identifiers are all distinct, but the weights are not required to be distinct.

We want to compute an approximation of a minimum spanning tree. A k-approximation of an MST is a spanning tree whose weight is not larger than k times the weight of an MST. Note that in our definition of approximation, we do not relax the requirement of acyclicity, thus it is not the same kind of approximation as the one used in the literature about spanners.

State model Our algorithm is designed for the classic *state model* [13]. In this model, every node has a state, and communication between neighbors is modeled by direct reading of states instead of exchanges of messages. This state is the *mutable memory*, that is, it 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 identifier and 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 value. Here by "local view" we mean the state of a node, the states of its neighbors, the identifier of the node, and the weights of the adjacent edges. If there exists a rule, such that the property of the node's view is satisfied, then we say that the node is *enabled*. (Note that for a node deciding whether it is enabled or not could take some time and space. As in most models in network distributed computing, such local computation is not taken into account for the time and space complexity.) The asynchrony of the system is modeled by a scheduler who 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. Other schedulers considered in the literature can have, for example, fairness constraints: they cannot activate always the same nodes. See [15] for a survey of the schedulers of the literature. The choice of the distributed unfair scheduler makes our algorithm the most robust possible.

To compute time complexities, we use the definition of *round* of [14]. 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 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* [23]. 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 as input the view of v, including the labels of v and of its neighbors, and decides whether to *accept* or 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 *not* satisfying P, there is *no* way for the prover to make all nodes accept. 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 starting from an arbitrary configuration, it reaches a correct configuration after some finite time, called the *stabilization time*, 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 [5], except that the proof is not given by an oracle: it is built by the algorithm. We refer to [18, 20] for surveys on distributed proofs.

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.

Two-phase approach In order to reach a configuration where the nodes can safely stop updating their states, we need first to have a correct solution at hand, and second to allow the nodes to be sure that indeed the solution is correct. As said earlier, the classic way to do this is to keep in memory some key extra pieces of information gathered during the computation. As it seems that this approach is difficult to implement here, we use another way: we first build the solution alone, and then we build a certification (*i.e.* a distributed proof) of this solution.

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 computation, starting from a clean configuration, we cannot end up in a configuration that is detected as incorrect. If this were to happen, then the scheduler could make the algorithm go through a reset infinitely often, and it would never 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 and how they work together 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, thus allows the silent stabilization (detailed in Section 7), and a reset algorithm that can erase everything and go back to a clean configuration. We describe the algorithm in a modular way to ease the reading, but in the end our result is one algorithm. In particular, it is important to have the three pieces working together. Note that such modular design for a self-stabilizing algorithms is not new, even for unfair schedulers: for example in the recent and celebrated coloring algorithm of [1] there is a first algorithm reducing the number of colors very fast and then a second algorithm to finish the job by eliminating the last extra-colors.

In our algorithm, the reset procedure is dominant, in the sense that if a reset is launched it will basically overrule the other procedures. For this reset we take a solution from the literature: Devismes and Johnen [12] recently proposed a cooperative (that is, tolerating multiple simultaneous initiators) silent self-stabilizing reset algorithm that satisfies our constraints in terms of scheduler, stabilization time and space complexity. Then to articulate the two other pieces, we simply use flags that indicate for each node which algorithm it is running. If the algorithm are not run one after the other, then there will be a local inconsistency between these flags, and the reset will be launched.

One last element about how the pieces work together is related to the weight transformation. As said earlier, a key ingredient to get our approximation algorithm in small space (Theorem 1) is a transformation of the weights. This transformation consists in replacing each weight by a smaller weight. But, as we have limited space, we cannot store the transformed weights of all the edges adjacent to a node in its state (there can be $n-1$ edges adjacent to a node). Instead, every time a step is taken, the node recomputes the new weights to know which rule applies.

4 Approximation-memory trade-off

In this section, we show how we can replace the original weights with approximated weights to decrease the number of bits used to encode them, while preserving guarantees on the MST computed. The precise trade-off between the space used for the new weights and the approximation is given in Lemma 1. As the expression is not very elegant, this is the only place of the paper where we write it explicitly. The trade-off for the whole algorithm can be derived from the values of this lemma: simply multiply by $O(\log n)$.

Lemma 1. *There exists a transformation of the weights that allows for a trade-off between space needed to encode the new weights, and the quality of the tree one can compute from them. More precisely, for every integer k in the range $[-\log\log n, \log n]$, we can get new weights with the following properties:*

- *for $k = \log n - 1$, approximation 1 and size $\log n + 1$,*
- *$k \in [0, \log n - 2]$,, approximation $1 + \frac{1}{2^k}$ and space $k + \log(\log n - k + 1)) + 1$,*

– *for* $k \in [-\log\log n, 0]$ *approximation* $2^{2^{-k}}$ *and space* $\log\left(\frac{\log n}{2^{-k}} + 1\right) + 1$.

Before proving the lemma, let us restate and prove the three points of the trade-offs that we have already mentioned.

Corollary 1. *The construction of Lemma 1 gives in particular:*

– *Exact solution, with weights encoded on* $O(\log n)$ *bits.*
– *2-approximation, with weights encoded on* $O(\log\log n)$ *bits.*
– *A trivial guarantee (poly*(n)*-approximation), with weights encoded on a constant number of bits.*

The exact solution directly follows from the case $k = \log n - 1$. The 2-approximation corresponds to $k = 0$. The arbitrary approximation corresponds to $k = -\log\log n$.

The full proof of Lemma 1 is deferred to the full version [7], but we give a sketch of the argument now. The explanation is illustrated by Figure 1.

The core idea of the weight transformation is to group the weights into buckets, that will be assigned the same new weight. The larger the buckets, 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 w, will be in a bucket b, if $2^{b-1} < w \leq 2^b$. This way, every weight is at most doubled, and an MST computed on these new weights is at most twice as heavy as the MST with the original weights. The good thing is that now there are $O(\log n)$ different weights instead of n, which means it can be encoded on $O(\log\log n)$ bits instead of $O(\log n)$. Now for various reasons explained in the full version [7], we do not use this vanilla version of exponential bucketing with other bases for other approximation ratios. Instead we consider the series of the rounding values 1, 2, 4, 8, ..., $2^{\log n}$ given by the technique above, that we call *milestones*, and work on it. We remove some of the milestones to get a coarser approximation or we add new ones to get a more fine-grain approximation.

5 MST construction algorithm

In this section, we give a distributed algorithm for minimum spanning tree. This algorithm is not self-stabilizing; the self-stabilizing part will be taken care of in Section 7. As our main goal is to use small space, we do not optimize the time of this construction algorithm (except that we want it to be polynomial), and keep it as simple as possible.

Lemma 2. *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

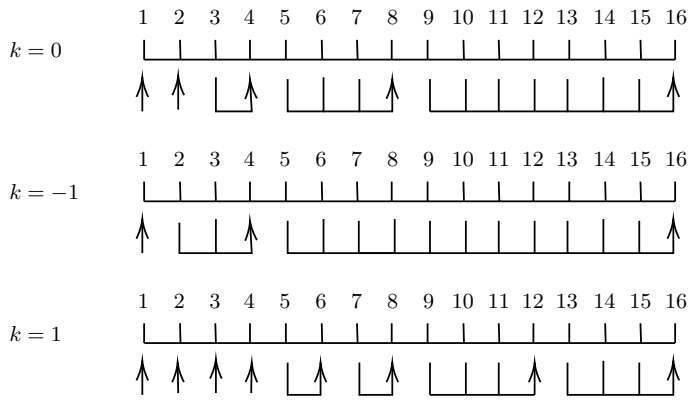


Fig. 1. Illustration of Lemma 1. For $k = 0$, for example the weight 2, stays as 2, but weight 3 is transformed into 4. For the case $k - 1$, we remove basically half milestones, and for $k = 1$ we basically double the number of milestones.

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 to 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 tree. The token transports 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 [6], inspired by the tree algorithm of [11] and the token algorithm of [24]. These algorithms ensure, under the distributed unfair scheduler, stabilization to a proper token circulation in $O(n)$ rounds, and they use only $O(\log n)$ bits of memory.

Finally, a node must be able to decide locally whether adding a specific edge would close a cycle or not. To do so every node will hold a *component name*. This name is the minimum identifier in the connected component of the node, in the current state of the tree. This way, a node can safely add an edge if its component name is different from the component name of the other endpoint. 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 in polynomial time and logarithmic memory [9].

The space complexity is in $O(s + \log n)$, because we just need to store a constant number of IDs, weights and additional $O(\log n)$-size objects. The time complexity in terms of rounds is polynomial. Indeed the number of phases is

polynomial, because we consider polynomial weights, and each phase lasts a polynomial number of rounds (because the primitives we use, token circulation and traversal of the tree, are known to be polynomial [6, 9, 11, 24]).

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 [22]. It is necessary here to describe the scheme of [22] with some details (even though it is not our work) in order to allow the reader to understand the next section without reading the full version in [22]. For an intuitive but more in depth description of the main ideas behind the scheme, we refer to [17].

Lemma 3 ([22]). *There exists a distributed proof of size $O(\log n \cdot s)$ for minimum spanning trees.*

Proof (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 be certified with $O(\log n)$ bits [23], 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 edges of the tree. It is rather easy to check that the correctness of these pieces of information can be checked locally.

To show why this is 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 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). First, remember that adding (u, v) would lead 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) (and thus that could be replaced by (u, v)). Therefore, u and v only have to look for such an edge. The path from u to v must go through the root, because these 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. As the nodes are given these maximum weights on the paths to the root in their labels, they can check whether (u, v) is lighter or not than the heaviest edge on the path.

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 (*i.e.* the center of its subtree of the first center), its third center (*i.e.* 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 [21]) 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. Surprisingly, this is enough to describe and check each orientation, and it takes only constant number of bits per level. (See [22] or [17], to see for example why not knowing the precise child-parent relation is not problematic.)

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 4. *Given a tree with an 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 copies, 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, therefore we cannot write the whole list in the state of the center. 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.* [9] 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, looking for the next center. Indeed we have a tree, with a root, with a proper orientation, and with correct subtree numbers.

Now, let us 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 cycle 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.

When the computations ends on all nodes, every node is labeled as specified in Section 6, and the algorithm becomes silent.

8 Conclusion

This paper presents the first self-stabilizing algorithm using approximation to reduce memory usage. A step in this construction is the design of a polynomial-time silent self-stabilizing algorithm for MST construction using $O(\log n \cdot s)$ bits of memory. This later algorithm uses an unusual two-phase approach: building and then certifying the solution. We believe that this modular approach is the key to go down to complexity $O(\log n \cdot s)$, and we think that it is an interesting problem to formally prove this intuition.[6]

[6] Some elements indicating that at least the classic techniques cannot avoid a modular approach can be be found in [19]

Bibliography

[1] Leonid Barenboim, Michael Elkin, and Uri Goldenberg. Locally-iterative distributed (Δ+ 1): -coloring below szegedy-vishwanathan barrier, and applications to self-stabilization and to restricted-bandwidth models. In Calvin Newport and Idit Keidar, editors, *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom, July 23-27, 2018*, pages 437–446. ACM, 2018.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] Lélia Blin, Swan Dubois, and Laurent Feuilloley. Silent MST approximation for tiny memory. *CoRR*, abs/1905.08565, 2019. URL `http://arxiv.org/abs/1905.08565`.

[8] Lélia Blin, Laurent Feuilloley, and Gabriel Le Bouder. Brief announcement: Memory lower bounds for self-stabilization. In Jukka Suomela, editor, *33rd International Symposium on Distributed Computing, DISC 2019, October 14-18, 2019, Budapest, Hungary*, volume 146 of *LIPIcs*, pages 37:1–37:3. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.DISC.2019.37. URL `https://doi.org/10.4230/LIPIcs.DISC.2019.37`.

[9] 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.

[10] Keren Censor-Hillel, Ami Paz, and Mor Perry. Approximate proof-labeling schemes. *Theor. Comput. Sci.*, 811:112–124, 2020. doi:10.1016/j.tcs.2018.08.020.

[11] 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.

[12] Stéphane Devismes and Colette Johnen. Self-stabilizing distributed cooperative reset. In *39th IEEE International Conference on Distributed Computing Systems, ICDCS 2019*, pages 379–389, 2019. doi:10.1109/ICDCS.2019.00045.

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

[14] 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.

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

[16] Yuval Emek and Yuval Gil. Twenty-two new approximate proof labeling schemes. In *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, volume 179, pages 20:1–20:14, 2020. doi:10.4230/LIPIcs.DISC.2020.20.

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

[18] Laurent Feuilloley. Introduction to local certification. *CoRR*, abs/1910.12747, 2019. URL `http://arxiv.org/abs/1910.12747`.

[19] Laurent Feuilloley. Can we always build and certify at the same time? (Maybe not). Discrete notes. `https://discrete-notes.github.io/build-certify-3`, 2020.

[20] Laurent Feuilloley and Pierre Fraigniaud. Survey of distributed decision. *Bulletin of the EATCS*, 119, 2016. url: bulletin.eatcs.org link, arXiv: 1606.04434.

[21] 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.

[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] 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.