Space-Optimal Time-Efficient Silent Self-Stabilizing Constructions of Constrained Spanning Trees

Lélia Blin* and Pierre Fraigniaud†

*LIP6-UPMC, University of Evry-Val d'Essonne, France
†CNRS and University Paris Diderot, France

***Abstract*—Self-stabilizing algorithms are distributed algorithms supporting transient failures. Starting from any configuration, they allow the system to detect whether the actual configuration is legal, and, if not, they allow the system to eventually reach a legal configuration. In the context of network computing, it is known that, for every task, there is a self-stabilizing algorithm solving that task, with optimal space-complexity, but converging in an exponential number of rounds. On the other hand, it is also known that, for every task, there is a self-stabilizing algorithm solving that task in a linear number of rounds, but with large space-complexity. It is however not known whether for every task there exists a self-stabilizing algorithm that is simultaneously space-efficient *and* time-efficient. In this paper, we make a first attempt for answering the question of whether such an efficient algorithm exists for every task, by focussing on constrained spanning tree construction tasks. We present a general roadmap for the design of silent space-optimal self-stabilizing algorithms solving such tasks, converging in polynomially many rounds under the unfair scheduler. By applying our roadmap to the task of constructing minimum-weight spanning tree (MST), and to the task of constructing minimum-degree spanning tree (MDST), we provide algorithms that outperform previously known algorithms designed and optimized specifically for solving each of these two tasks.**

I. Introduction

A. General Context

Self-stabilization [27] deals with the design and analysis of distributed algorithms in which processes are subject to *transient* failures modifying the content of their variables. The main objective of self-stabilization is to evaluate the capacity for an asynchronous distributed system to recover from transient faults, that is, to measure the ability of the system to return to a legal state starting from an arbitrary state, and to remain in legal states whenever starting from a legal state. The legality of a state is a notion that depends on the task to be solved, like, e.g., the presence of a unique leader, in the case of the leader election task.

In the context of network computing, each process is a node of a network modeled by an n-node graph G. The nodes act asynchronously, and they communicate along the edges of G. More specifically, in the *state model* [26], every node has read/write access to its own variables, and read-only access to the variables of its neighbors in G. In one atomic *step*, a node performs the following three successive actions: (1) read its variables and the variables of its neighbors, (2) perform imdividual computation, and (3) update the content of its variables. A process that aims at performing a step is said *activatable*, and which activatable node actually performs a step is under the control of a *scheduler*. A *round* of an algorithm starting from some initial state s in some execution $\mathcal{E}$ is the shortest prefix of $\mathcal{E}$ in which each activatable process in s performs at least one step or becomes non activatable due to the actions of one of its neighbors.

A property that is often considered desirable for a self-stabilizing algorithm is to be *silent* [28], which is a form of *termination*. Specifically, a silent algorithm is an algorithm which satisfies that, once a legal state has been reached, the individual state of each process remains unchanged. That is, a silent algorithm insures that the system converges to some legal state, and then the processes "terminate" in the sense that they do not modify their variables anymore, unless some faults occur. Of course, the processes remain active, and perpetually check their variables and the variables of their neighbors since, if a fault occurs, the algorithm must be able to detect it (in order to proceed in a way enabling the system to return to a legal state).

Designing silent algorithms is difficult because one must insure that the processes are able to collectively decide *locally* of the legality of a (global) state of the system, based solely on their own individual states and on the individual states of their neighbors. This difficulty becomes prominent when one takes into account an important complexity measure for self-stabilizing algorithms: *space complexity*, i.e., the amount of memory used at each process to store its variables [11], [28]. Keeping the memory space limited at each process reduces the potential corruption of the memory, and enables to maintain several redundant copies of the variables (e.g., for fault-tolerance) without hurting the efficiency of the system. Moreover, in the state model, keeping the space complexity small insures that reading variables in registers of neighboring processes does not consume too much bandwidth.

B. Objective

It was recently established (see [15]) that, in the state model, for every task, there is a silent self-stabilizing algorithm solving that task, with optimal space-complexity. However, this algorithm is converging in a number of rounds exponential in the size of the network. On the other hand, [15] also shows

The first author received additional supports from ANR project IRIS, and the second author from ANR project DISPLEXITY, and INRIA project GANG.

that, for every task, there is a silent self-stabilizing algorithm solving that task in a linear number of rounds. However, this latter algorithm reaches that performance to the cost of a large space-complexity, potentially exponentially larger than the lower bound for silent algorithms. It is not known whether one can get the best of both worlds, and [15] explicitly rose the following question:

Problem 1.1: Is there, for every task, a (silent) self-stabilizing algorithm for solving that task, that is simultaneously space-efficient and time-efficient?

In the spirit of [15], space-efficiency and time-efficiency respectively refer to a space complexity of the same order of magnitude as the most compact *proof-labeling scheme* [52] for the considered task, and to a number of rounds polynomial in the number of nodes. Recall that a proof-labeling scheme for a task is an assignment of short *labels* to the nodes such that, provided with these labels, the nodes can collectively verify the legality of a system configuration[1] by inspecting the variables and the labels in their vicinities at distance 1 in the network. Other fault detectors and local monitoring mechanisms have been proposed in the literature (see, e.g., [2], [10]). Nevertheless, this paper sticks to proof-labeling scheme for consistency with [15].

Our objective is to tackle Problem 1.1, at least for a wide class of tasks. More precisely, the paper focusses on the self-stabilizing construction of various kinds of *constrained spanning trees* in networks. Such a task is described by a family $\mathcal{F}$ of spanning trees, and the task consists in, given a network G, constructing a spanning tree $T \in \mathcal{F}$ of G. Typically, the tree T is rooted at some node r, and it is distributedly encoded at each node $v \neq r$ by storing the identify of v's parent $p(v)$ in T, while the root r has $p(r) = \perp$. Typical families of constrained spanning trees are breadth-first search trees (BFS), minimum-weight spanning trees (MST), or minimum-degree spanning trees (MDST).

C. Contributions of the Paper

In a nutshell, we prove that, for a wide class of constrained spanning tree construction tasks, the answer to Problem 1.1 is positive (cf. Theorems 3.1 and 7.1). While this result does not fully solve that problem, it provides hope for answering Problem 1.1 positively for large batches of natural tasks.

More specifically, we say that a family $\mathcal{F}$ of spanning trees admits a *local search* algorithm if, for every $T \notin \mathcal{F}$, there exists a fundamental cycle[2] $T + e$ such that switching a tree edge f in this fundamental cycle with the non-tree edge e results in a tree $T' = T + e - f$ that is "closer" to $\mathcal{F}$, where the distance to $\mathcal{F}$ is measured according to some potential function. A family $\mathcal{F}$ is then said *efficiently locally searchable* if it admits a local search algorithm for which the distance to $\mathcal{F}$ is based on an appropriate assignment of short *labels* to the nodes (as short as the size of the most compact proof-labeling scheme for $\mathcal{F}$), such that both constructing the labels, and finding a fundamental cycle enabling some improvements of the current tree can be done in a polynomial number of rounds, using small memory at each node. We establish that, for every efficiently locally searchable family $\mathcal{F}$ of spanning trees, there exists a silent self-stabilizing construction algorithm for $\mathcal{F}$ converging in *polynomially many rounds*, with *almost optimal space complexity* $O(k + \log n)$, where k denotes the minimum space complexity of a proof-labeling scheme for $\mathcal{F}$ (which is known to be a lower bound on the space-complexity for silent algorithms [15], [52]).

The above general result has several corollaries. For instance, we can infer from it that there exists a silent self-stabilizing construction algorithm for MST, using $O(\log^2 n)$-bits memory, and converging in poly(n) rounds. This MST algorithm has an optimal space-complexity, as a consequence of the lower bound in [50]. While there exist more compact MST algorithms (see, e.g., [17], [51]), these latter algorithms, specifically designed for minimizing the size of the memory, are not silent.

By extending our notion of efficiently locally searchable families of spanning trees for allowing operations of the type $T \leftarrow T \cup \{e_1, \ldots, e_k\} \setminus \{f_1, \ldots, f_k\}$, where $e_i \notin T$ and $f_i \in T$ are specific kinds of "nested" edges, $i = 1, \ldots, k$, we can infer a perhaps more significant corollary from our general result. We prove that there exists a silent self-stabilizing construction algorithm for MDST, stabilizing on a class of spanning trees with degree at most $\text{OPT} + 1$, and using optimal-size $O(\log n)$-bits memory. The algorithm converges in poly(n) rounds, and performs polynomial-time computations at each node. This MDST algorithm is an additive approximation algorithm. It returns spanning trees with degree at most $\text{OPT} + 1$. It uses registers of $O(\log n)$ bits, which is optimal as a consequence of the lower bound in [28]. It exponentially improves the previous best known $(\text{OPT} + 1)$-approximation algorithm [16], which is not silent, yet is using $\Omega(n \log n)$ bits of memory per node. In fact, our MDST algorithm constructs a special kind of trees, named *FR-trees* after Fürer and Raghavachari [33]. Indeed, we show that verifying whether a given tree is an arbitrary tree of degree $\leq \text{OPT} + 1$ cannot be done in polynomial time, unless NP = co-NP. Instead, we show that there is a proof-labeling scheme for FR-trees using labels on $O(\log n)$ bits.

Achieving these results relies on a collection of ingredients, each of them having its interest on its own. The first ingredient is based on the original concept of *malleable* proof-labeling scheme, defined as a proof-labeling scheme which does not rise an alarm when a legal solution is modified into another "close" legal solution. We prove that there exists a malleable proof-labeling scheme for spanning trees. Such a scheme allows us to design a silent *loop-free* self-stabilizing algorithm for permuting tree edges with non-tree edges in a spanning tree.

The second ingredient is the design of a silent self-stabilizing algorithm for providing the nodes with the $O(\log n)$-bit labels corresponding to the informative labeling scheme from [6], designed to compute the *nearest common*

[1] If the configuration is legal then all nodes must accept it, otherwise at least one node must reject it.

[2] Given a spanning tree T and a non-tree edge $e = \{u, v\}$, recall that the *fundamental cycle* $T + e$ is the cycle formed by e and the simple path in T between its two extremities u and v.

ancestor (NCA) of any two nodes in a tree. More specifically, thanks to this informative-labeling scheme, given any two nodes u and v in a tree T, the NCA of u and v can be computed based solely on the labels $\lambda(u)$ and $\lambda(v)$ of nodes u and v. Our silent self-stabilizing implementation of the NCA-labeling scheme [6] is used for identifying the fundamental cycles in a tree. Up to our knowledge, this is the first time that this very compact NCA-labeling scheme is used in the context of self-stabilization. Moreover, in order to use this scheme as a subroutine for our silent self-stabilizing algorithm, we have designed a proof-labeling scheme for that scheme. It is probably the first occurrence of a proof-labeling scheme for an informative-labeling scheme!

D. Related work

Our paper is directly inspired from [15], which established the following two complementary results: (1) For every task $\mathcal{T}$, there exists a silent self-stabilizing algorithm solving $\mathcal{T}$ in $O(n2^n)$ rounds, using at most $O(\text{OPT}+\log n)$ bits of memory; (2) For every task $\mathcal{T}$, there exists a silent self-stabilizing algorithm solving $\mathcal{T}$ in $O(n)$ rounds, using $O(n^2)$ bits of (public) memory. In the same paper, the authors ask the question of whether, for every task $\mathcal{T}$, there exists a silent self-stabilizing algorithm solving $\mathcal{T}$ in $O(\text{poly}(n))$ rounds while using at most $O(\text{OPT} + \log n)$ bits of memory.

There is a huge literature on the self-stabilizing construction of various kinds of trees, including spanning trees (ST) [20], [22], [53], breadth-first search (BFS) trees [1], [3], [18], [24], [30], [42], [48], depth-first search (DFS) trees [21], [23], [24], [43], minimum-weight spanning trees (MST) [13], [17], [39], [41], [51], shortest-path spanning trees [38], [44], minimum-diameter spanning trees [12], minimum-degree spanning trees (MDST) [16], etc. Some of these constructions are even silent, with optimal space-complexity. This is for instance the case of several BFS constructions under different kinds of schedulers [3], [18], [24], [42], and of the ST constructions in [22], [53].

More specifically, regarding MST, [17], [51] have proposed compact self-stabilizing constructions, using just $O(\log n)$ memory per node. These compact algorithms are however not silent. ([51] is uniform and converges in $O(n)$ rounds, while [17] is just semi-uniform, and converges in $O(n^3)$ rounds). [50] proves that, for being silent, any algorithm requires registers on $\Omega(\log^2 n)$ bits. Proof-labeling schemes matching this bound can be found in [50] and [52]. In this paper, we show that, by plugging these schemes into our general framework, we can get a polynomial-time silent self-stabilizing algorithm also matching this bound the on memory space.

Recall that, for any given (connected) graph G, a spanning tree T of G is a minimum-degree spanning tree (MDST) if its degree is minimum among all spanning trees of G. Our interest in MDSTs was originally motivated by resolving issues arising in the design of MAC protocols for sensor networks under the 802.15.4 specification [59]. Nevertheless, is also worth pointing out that MDSTs arise in many other contexts, including electrical circuits [54], communication networks [31], as well as in many other areas [36], [45]. There is, up to our knowledge, only one known self-stabilizing algorithm for MDST construction, in [16]. This algorithm is an $(\text{OPT}+1)$-approximation. It is not silent, and requires nodes to be granted with a memory as large as $\Omega(n \log n)$ bits. Prior to our work, there were no known proof-labeling schemes for MDST. In this paper, we design a $O(\log n)$-bit proof-labeling scheme for a subclass of near optimal MDST, and show that, by plugging this scheme into our general framework, we can get a polynomial-time silent self-stabilizing algorithm for approximating MDST within $+1$ from the optimal, using registers on $O(\log n)$ bits.

In addition to these references, there is a series of contributions that are closely related to our work. In particular, several papers address the leader election task [7], [8], [11], [25], [29], which is inherently related to spanning tree construction. Regarding the sequential construction of minimum-degree spanning trees, the best known result is [33] which describes a polynomial-time algorithm for constructing spanning as well as Steiner trees with degree at most $\text{OPT} + 1$. Several generalizations of the minimum-degree spanning tree problem have been addressed, including the degree-bounded minimum-weight spanning tree problem [58], and minimum-degree spanning tree problem in digraphs [49].

II. Model and objective

A. Self-stabilization

In this paper, we are dealing with the *state model* for self-stabilization (see [26]), where each process is a node of an asynchronous network modeled as a simple connected graph $G = (V, E)$ with n nodes. Every node is a state machine with state-set S (the same for all nodes). Each node $v \in V$ has a distinct identity, denoted by $\text{ID}(v) \in \{1, \ldots, n^c\}$ for some constant $c \geq 1$. Every node is aware of its identity, which is a (non corruptible) constant. Similarly, in an edge-weighted graph G, every node v is aware of the weights of its incident edges, which are (non corruptible) constants. As for node identities, we make the classical assumption that all weights can be stored on $O(\log n)$ bits. Moreover, w.l.o.g., we assume that all weights are pairwise distinct [34].

In the state model, each node v has read/write access to a single-writer multiple-reader register which stores the current state of v. Moreover, in one *atomic* step, every node executes three successive operations, consisting in: (1) reading its own register, and the registers of its neighbors in G, (2) applying a transition function on these data, and (3) writing the result of this application into its register. The transition function is a function $\delta : S^* \to S$ which, given any finite collection of states, returns a new state. At node v in state s_0, and given the states $s_1, \ldots, s_d$ of the d neighbors of v in G, the new state s'_0 of v after one step is $s'_0 = \delta(s_0, \{s_1, \ldots, s_d\})$. The network is asynchronous in the sense that nodes take step of computation (i.e., change state) in arbitrary order, under the control of a *scheduler*. We consider the most liberal setting by assuming the *unfair* scheduler. That is, at each step, the scheduler is only bounded to choose at least one of the *enabled* or *activatable* node (those for which the algorithm aims at taking a step). A collection of n individual register states in an n-node graph

form a (global) *state* of the system. As in [15], a *task* is specified by a set of states, called *legal* states. For instance, in the case of the (unconstrained) spanning tree construction task, each node v maintains a variable $p(v)$ storing either the identity of its parent, or $\perp$. A state is then legal if and only if the 1-factor defined by the set of (directed) edges $\{(v, p(v)), v \in V\}$ form a spanning tree of G.

A *fault* is the corruption of the register of one or more nodes in the system. After a fault has occurred, the system may be in an illegal state. Note that, for variables storing information whose size may vary depending on the structure of the network, like, typically, the identity of a leader, the corruption of that variable cannot result is storing a value with arbitrary large size. Moreover, recall that node identities and edge-weights cannot be corrupted. (Instead, variables storing these values in registers can be corrupted). It is the role of the self-stabilizing algorithm to detect the illegality of the current state, and to make sure that the system returns to a legal state. In other words, starting from any state, the system must eventually converge to a legal one, and must remain in legal ones. A self-stabilizing algorithm is *silent* if and only if it converges to a legal state where the values of the registers used by the algorithm remain fixed.

Given a state γ of the system, let $X \in V$ be the set of enabled nodes in γ. A *round* of an execution $\mathcal{E}$ of an algorithm $\mathcal{A}$ starting from γ is the shortest prefix of $\mathcal{E}$ in which each node in X executes at least one step, or becomes non activatable due to the actions of one of its neighbors. If $\mathcal{A}$ constructs and stabilizes on states in some family F of states, then the round-complexity of $\mathcal{A}$ is the maximum, taken over all initial states γ, and over all executions $\mathcal{E}$ of $\mathcal{A}$ starting from γ and ending in a state $\gamma' \in F$, of the number of rounds in $\mathcal{E}$. The latter is the integer k such that $\mathcal{E}$ can be decomposed in a sequence $\gamma_0 = \gamma, \gamma_1, \ldots, \gamma_k = \gamma'$ such that, for every $i = 0, \ldots, k-1$, the round of $\mathcal{E}$ starting from γ_i ends in γ_{i+1}.

B. Constrained spanning trees construction tasks

Let $\mathcal{F}$ be a family of trees such that, for every connected graph G, there exists a spanning tree T of G with $T \in \mathcal{F}$. To describe $\mathcal{F}$, it is sufficient to define the set $\mathcal{F}(G)$ of spanning trees of G belonging to $\mathcal{F}$. Typical examples of such a family are $\text{ST}(G) = \{T : T \text{ is a spanning tree of } G\}$, and $\text{MST}(G) = \{T : T \text{ is a minimum-weight spanning tree of } G\}$. Given a tree T, the *degree* $\deg(T)$ of T is the maximum, taken over all nodes v of T, of the degree of v in T. Given a graph $G = (V, E)$, a spanning tree T of G is of minimum degree if there are no spanning trees T' of G with $\deg(T') < \deg(T)$. We denote by $\Delta_{min}(G)$ the degree of any minimum-degree spanning tree of G. Since deciding whether a graph is Hamiltonian is NP-hard, we get that deciding, given G and $k \geq 0$, whether $\Delta_{min}(G) \leq k$ is NP-hard. However, thanks to the (sequential) algorithm by Fürer and Raghavachari [33], given any graph G, one can construct a spanning tree T of G with $\deg(T) \leq \Delta_{min}(G) + 1$, in polynomial time. Hence, we are interested in the family near-$\text{MDST}(G) = \{T : T \text{ is a spanning tree of } G, \text{ and } \deg(T) \leq \Delta_{min}(G) + 1\}$.

This paper describes self-stabilizing distributed algorithms for different families $\mathcal{F}$ of spanning trees. Given a family $\mathcal{F}$, the algorithm for $\mathcal{F}$ running in a network G must return a tree $T \in \mathcal{F}(G)$. This spanning tree is encoded distributedly as follows: it is rooted at an arbitrary node r, and every node $v \in V(G)$ stores the identity $p(v)$ of its parent in T (the root r stores $p(r) = \perp$). When the algorithm stabilizes, the underlying graph of the 1-factor $\{(v, p(v)) : v \in V(G) \text{ and } p(v) \neq \perp\}$ must be in $\mathcal{F}(G)$.

C. Proof-labeling schemes

A silent algorithm needs to detect locally, by having each node inspecting only its register and the registers of its neighbors, whether the current state of the system is legal or not. A typical mechanism for doing so is *proof-labeling scheme*, defined by a prover-verifier pair (p, v). The *prover* p is charge of assigning a *label* (i.e., a bit-string) $\lambda(v)$ to every node v so that, given their own labels, and the labels of their neighbors, the nodes can collectively decide, by applying the local *verifier* v at each node, whether or not the current configuration of the system satisfies a given graph property. If this property holds, then all nodes must accept the current configuration (e.g., output "yes"), otherwise at least one node must reject it (e.g., output "no"). More precisely, if the property is satisfied, then there must exists a label-assignment to the nodes (provided by the prover) such that the verifier accepts at every node. And, conversely, if the property is not satisfied, then, for every label-assignment to the nodes, the verifier must reject in at least one node.

For instance, the following proof-labeling scheme for ST, the family of all spanning trees, is known for long (see, e.g., [47]). The label $\lambda(v)$ assigned by the prover to node v in a spanning tree T is a pair (ID, d) where ID is the identity of the root r of T, and d is the distance in T (i.e., number of hops) between v and r. At every node v, the verifier checks that the given root-identity ID is identical to the root-identity given to all its neighbors in G, and checks that the distance given to its parent $p(v)$ is one less than the distance d given to v (the root r checks that $d = 0$). We call this scheme *distance-based*. It is easy to see that if T is not a spanning tree of G, that is, if T is not spanning all nodes of G, or if T is not a tree (T may be a forest, or T may contain a cycle), then some inconsistencies will be detected at some node(s), for every given collection $\{\lambda(v), v \in V(G)\}$ of labels. The distance-based scheme uses labels on $O(\log n)$ bits, and the verification performed at each node runs in polynomial time – it merely consists of $k + 1 = O(n)$ integer comparisons at each node of degree k.

III. Spanning tree constructions guided by proof-labeling schemes

Let $\mathcal{F}$ be a family of spanning trees. We aim at defining a potential function ϕ which, given any graph G, and any spanning tree T of G, returns a non-negative value $\phi(T)$ measuring how close the spanning tree T is from being in $\mathcal{F}$. We show that this can be achieved using a proof-labeling scheme (p, v) for $\mathcal{F}$ by setting up ϕ so as to measure how much T violates the correctness of the scheme. More precisely, we aim at defining a function ϕ that satisfies (1) $\phi(T) \geq 0$ for

every spanning tree T of any graph G, and (2) $\phi(T) = 0$ if and only if $T \in \mathcal{F}$. We say that a function ϕ is *cyclical-decreasing* if it satisfies (1) and (2), plus a third condition stating that if $\phi(T) > 0$, then there must exist a fundamental cycle of $T + e$ for some non-tree edge e, and an edge $f \in T$ of that cycle, such that $\phi(T+e-f) < \phi(T)$. A cyclical-decreasing function ϕ trivially yields a sequential local-search greedy algorithm for constructing a spanning tree $T \in \mathcal{F}$ of any given graph G. We call this algorithm *proof-labeling scheme guided* (or *PLS-guided* for short) spanning tree construction, as depicted below:

Algorithm 1 *PLS-guided spanning tree construction I*

Require: a graph G, and a proof-labeling scheme (p, v) for a family $\mathcal{F}$ of spanning trees
1: construct a spanning tree T of G
2: **while** $\phi(T) \neq 0$ **do**
3: find edges e and f such that $\phi(T + e - f) < \phi(T)$
4: $T \leftarrow T + e - f$
5: **end while**
6: output T

Implementing such an algorithm in a distributed silent self-stabilizing manner requires to solve three important issues, beside the fact that the algorithm must construct and maintain a tree. (Instruction 1 can be implemented using, e.g., the algorithm in [25]). One issue is to design a mechanism enabling to set up the labels in the current spanning tree T, fitting with the value of ϕ for this spanning tree. Note that this mechanism may not recompute all labels from scratch at each iteration, but may just update the current labels in the updated tree. Note also that this mechanism does not need to compute $\phi(T)$, but must just make sure that at least one node notices that $\phi(T) \neq 0$. Second, the algorithm must be able to find a fundamental cycle $T+e$, and an edge $f \in T$ in this cycle such that $\phi(T + e - f) < \phi(T)$. Third, the algorithm must include a mechanism enabling to switch the two given edges e and f for producing the new tree.

The first issue (i.e., setting up the labels) is problem dependent. The second issue (i.e., finding the edges e and f) is also problem-dependent, but only partially. Indeed, the most crucial part is to come up with a mechanism enabling to identify and manipulate fundamental cycles, which is not problem dependent. The third issue (i.e., switching edges) is problem independent per se. Hence, in sections IV and V, we respectively describe how to switch edges, and how to handle fundamental cycles, in a silent self-stabilizing manner. This will enable us to establish the following results.

Let t_{label} and s_{label} be the number of rounds, and the space complexity, respectively, for setting up the labels in trees, in accordance to ϕ, in a silent self-stabilizing manner. Similarly, let t_{find} and s_{find} be the number of rounds and space complexity for finding the edges $e \notin T$ and $f \in T$ to be switched, in a silent self-stabilizing manner. Finally, let ϕ_{max} be the maximum value of ϕ.

Lemma 3.1: If a family $\mathcal{F}$ of spanning trees admits a cyclical-decreasing potential function ϕ, then there is a silent self-stabilizing implementation of Algorithm 1 solving task $\mathcal{F}$ with time complexity $\phi_{max} \cdot (t_{label} + t_{find} + O(n))$ rounds, and with registers of size $s_{label} + s_{find} + O(\log n)$ bits.

An important consequence of this lemma is that we can answer positively to Problem 1.1 for every spanning tree construction task $\mathcal{F}$ admitting a cyclical-decreasing potential function ϕ, and satisfying that (1) s_{label} is equal to the size of the most compact proof-labeling scheme for $\mathcal{F}$, with t_{label} polynomial in n, (2) $s_{find} = O(\log n)$ bits, with t_{find} polynomial in n, even if one is restricted to handling fundamental cycles encoded on $O(\log n)$ bits using the NCA-labeling scheme defined in [6], and (3) ϕ_{max} grows at most polynomially with n. Such a family $\mathcal{F}$ of trees is said to be *efficiently locally searchable*.

Theorem 3.1: For every efficiently locally searchable family $\mathcal{F}$ of spanning trees, there is a silent self-stabilizing implementation of Algorithm 1 solving task $\mathcal{F}$ in $\text{poly}(n)$ *rounds, and registers of size* $\text{OPT} + O(\log n)$ *bits.*

As we shall see further in the text, many common families of spanning trees are efficiently locally searchable. One trivial example is the families of all spanning trees. A simple, but less trivial example of efficiently locally searchable family of spanning trees is the family of BFS trees, exemplified below. (In Section VI, we show that MSTs are efficiently locally searchable).

Example: BFS construction. For a tree T rooted at r, let us provide each node u with its distance $d(u)$ to r in T. This essentially provides the family $\mathcal{F}$ of BFS trees with a proof-labeling scheme. The verifier at each node u essentially checks that non of its neighbors in the graph has a label smaller than $d(u) - 1$, and rejects the tree iff there is such a neighbor. For a spanning tree T with the distances correctly assigned, we set $\phi(T) = \sum_u |d(u) - \text{dist}_G(u, r)|$. We have $\phi(T) \geq 0$ for every spanning tree, and $\phi(T) = 0$ iff T is a BFS tree. Let us consider a node u that rejects the tree, and a neighbor v of u causing that rejection, with $d(v) < d(u) - 1$. In this case, the identifications of e and f are trivial: $e = \{u, v\}$ and $f = \{u, p(u)\}$. Switching e and f, i.e., resetting $p(u) = v$, and relabeling the subtree rooted at u with the appropriate distances in the new tree, yield $\phi(T + e - f) < \phi(T)$, and thus ϕ is cyclical-decreasing. Setting up the labels just requires $t_{label} = O(n)$ rounds, and $s_{label} = O(\log n)$ bits, by propagating a wave of distance assignments from the root of the tree to its leaves. Selecting the pair (e, f) of edges to be switched also trivially requires $t_{find} = O(n)$ rounds, and $s_{find} = O(\log n)$ bits, by letting the root of the tree do the selection in case many pairs are candidates. Finally, we have $\phi_{max} = O(n^2)$. As a consequence of Lemma 3.1, we get a polynomial-time space-optimal silent self-stabilizing BFS construction. (There are faster ad hoc BFS construction algorithms [25] — the objective of the paper is not to fight systematically for time optimality, but to show that the question risen in Problem 1.1 can be answered affirmatively for a large class of tasks, in the same spirit as interpreting P as the class of problems that can be sequentially solved efficiently).

IV. Proof of Lemma 3.1: A silent loop-free algorithm for switching edges

In this section, we sketch the proof of Lemma 3.1, for which it is essentially sufficient to show how to perform Instruction 4 of Algorithm 1 in a silent self-stabilizing manner, in $O(n)$ rounds, using $O(\log n)$ bits of memory per node. That is, we aim at designing an algorithm performing $T \leftarrow T + e - f$ for some given edges $e \notin T$, and f in the fundamental cycle of $T + e$, in a *loop-free* manner (i.e., always making sure that the current structure is a spanning tree). For this perspective, we introduce the novel notion of *malleable* proof-labeling scheme.

Definition 4.1: A proof-labeling scheme for a family $\mathcal{F}$ of data-structures is malleable *with respect to a transformation $\tau : \mathcal{F} \to \mathcal{F}$ if, for every $D \in \mathcal{F}$, there exists a correct labeling λ for D such that a correct labeling λ' for $D' = \tau(D)$ can be obtained from λ by replacing some entries in the labels of λ by $\perp$ (the discard symbol).*

We define a malleable proof-labeling scheme (p, v) for the family $\mathcal{F}$ of all spanning trees, with respect to the transformation $\tau(T) = T + e - f$ where $e \notin T$, and f is in the fundamental cycle of $T + e$. The label $\lambda(v)$ of node v provided by the prover p is a triple (ID, d, s) where, as in the aforementioned distance-based proof-labeling scheme for spanning tree (cf. Section II-C), ID denotes the identity of the root, and d denotes the distance to this root. The new parameter s denotes the *size* of the subtree rooted at v in the current tree. Hence, in particular, we must have $s_v = 1 + \sum_{u \in \text{child}(v)} s_u$ where child(v) denotes the set of children of v in the tree. As for the distance-based labeling scheme using the pairs (ID, d), the pairs (ID, s) alone provide a proof-labeling scheme for spanning trees. We call this latter labeling the *size-based* labeling scheme. So, let T be a spanning tree of G. Assign to each node v the due label $\lambda(v) = (d, s)$ corresponding to a distance-based labeling scheme, and to a size-based labeling scheme for T, with the same identity of the root (that we omit for the sake of simplifying the notations). This labeling is called *redundant*. We show that the redundant labeling scheme for spanning trees is malleable. For this purpose, we first define an appropriate way of *pruning* this labeling, by letting some distance and size variables unspecified (i.e., equal to $\perp$). Pruning however forbids creating pairs $(d, s) = (\perp, \perp)$. In addition, the following two constraints must be satisfied at every node v, where $\lambda(v) = (d, s)$, $\lambda(p(v)) = (d', s')$, and $\lambda'(\cdot)$ denotes the pruned labeling:

(C1) $\lambda'(v) = (d, \perp) \Rightarrow \lambda'(p(v)) = (d', \perp)$;
(C2) $\lambda'(v) = (\perp, s) \Rightarrow \lambda'(p(v)) = (d', s')$ or $(\perp, s')$.

Lemma 4.1: There exists a verification procedure v *satisfying the following two properties.* (1) *For any pruning of any legal redundant labeling of any spanning tree T, all nodes accept T.* (2) *For any labeling of a non-tree H with triples (ID, d, s) where d and s can potentially be $\perp$, at least one node rejects H.*

Proof: We consider the verification procedure described in the table below where "distance" stands for checking whether $d_v = d_{p(v)} + 1$, and "size" stands for checking whether $s_v = 1 + \sum_{u \in \text{child}(v)} s_u$, and output yes or no accordingly. (Of course, the presence of a unique root ID is also checked in all cases, as in the classical distance-based and size-based labeling schemes).

	Label of $p(v)$		
Label of v	(d', s')	$(d', \perp)$	$(\perp, s')$
(d, s)	distance and size	distance	size
$(d, \perp)$	no	distance	no
$(\perp, s)$	size	no	size

Hence, in particular, if both the label of v and the label of $p(v)$ are intact (i.e., no entries have been turned to $\perp$), then the verification performs at v by checking the distance property between v and $p(v)$, and the size property between v and all nodes in child(v). Instead, if the label of v is intact, but the label of $p(v)$ is of the form $(\perp, s)$, then the verification performs at v by checking the size property between v and its children.

Let T be a spanning tree with nodes labeled by pruning a legal redundant labeling. None of the labels can be of the form $(\perp, \perp)$ and the pruning must satisfy C1 and C2. If v has a label of the form (d, s), and $p(v)$ has a label of either the form (d', s') or the form $(d', \perp)$), then v outputs yes since the distance-based labeling is correct, and, whenever the pruned label of v is of the form (d, s) then, by C2, every child of v has label either (d'', s'') or $(\perp, s'')$. The same holds if v has a label of the form $(d, \perp)$, and $p(v)$ has a label of the form $(d', \perp)$. If v has a label of the form (d, s), but $p(v)$ has a label of the form $(\perp, s')$, then node v checks size. By C1 and C2, all the children of v have labels of the form (d'', s''), or $(\perp, s'')$. Thus v outputs yes since the size-based labeling is correct. Finally, if v has a label of the form $(\perp, s)$, then again it performs check size. By C2, all children of v have labels of the form (d'', s'') or $(\perp, s'')$. Thus v outputs yes since the size-based labeling is correct. As a consequence, the verification accepts the tree T, as claimed.

Conversely, let $H = \{(v, p(v)), v \in V \text{ and } p(v) \neq \perp\}$ be the 1-factor induced by the parent pointers, and assume that H is not a spanning tree of G. Assume, for the purpose of contradiction, that the verification accepts H. As a consequence, all nodes have the same value for the root ID in theirs labels. Therefore H contains a cycle C. The presence on C of a node with pruned label of the form $(d, \perp)$ implies that all nodes of C are of the form $(d, \perp)$, by C1. Hence, all nodes of C are either of the form (d, s) or $(\perp, s)$. In that case, the size-based labeling detects the cycle, a contradiction. Therefore, we conclude that the verification rejects H, as claimed. ∎

In order to replace an edge f on a fundamental cycle by a non tree-edge e, as on Fig. 1(a), one proceeds by a sequence of "local" switches between $\{v, u_{k-1}\}$ and $e = \{v, w'\}$, then between $\{u_{k-1}, u_{k-2}\}$ and $\{u_{k-1}, v\}$, and so on by performing successively the switches between $\{u_i, u_{i-1}\}$ and $\{u_i, u_{i+1}\}$ until the last switch between $f = \{u_1, w\}$ and $\{u_1, u_2\}$ which eventually removes f. In order to perform a local switch, such as replacing $\{v, w\}$ by $\{v, w'\}$ as on Fig. 1(b), we proceed in three phases, as detailed in [14]. In the *pruning* phase, node v initiated three "waves" of updates for the redundant labeling, resulting in a pruned labeling, as illustrated on Fig. 1(b): labels are pruned from (d, s) to $(d, \perp)$,

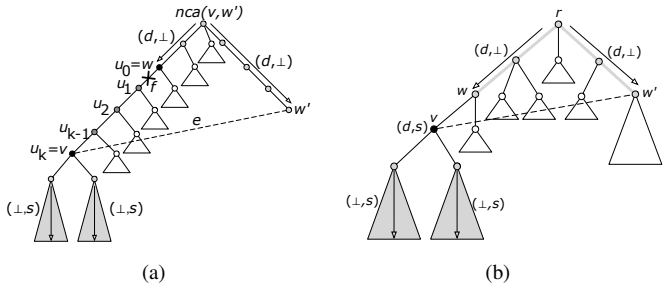

Fig. 1. *Switching tree-edges with non tree-edges*

downward along the paths $P_{r,w}$ and $P_{r,w'}$ from the root r of T to w and w', and are pruned from (d, s) to $(\perp, s)$ downward in the subtrees of v. Once the labels of w and w' have both be pruned to a label of the form $(d, \perp)$, and all children of v have their labels turned to $(\perp, s)$, the *switching* phase starts, where node v sets it parent to w', i.e., we now have $p(v) = w'$. Simultaneously, node v updates its distance to $1 + \text{dist}(w', r)$. Finally, during the *relabeling* phase, the former parent w of v recomputes the size of its subtree, by adding the sizes of all its children. Every node along $P_{r,w}$ proceeds the same successively, upward, up to r. Similarly, the new parent w' of v recomputes the size of its subtree, by adding the sizes of all its children. Then every node along the path $P_{r,w'}$ from w' to r proceeds the same successively, upward, up to r. Once v has changed its parent to w', every node in the subtree T_v of v recomputes its distance to the root, successively downward, down to the leaves.

By Lemma 4.1, non of the above manipulations on the labels of the nodes lead any node to reject the current data structure during the switch from one tree to another (i.e., the redundant labeling for spanning trees is malleable). Overall, the operation $T \leftarrow T - f + e$ performed as described above takes $O(n)$ rounds with $O(\log n)$ bits of memory per node (see [14]).

V. Proof of Theorem 3.1: Handling fundamental cycles with $O(\log n)$ bits

In this section, we sketch the proof of Theorem 3.1. To establish this result, it is essentially sufficient to prove that, given a non-tree edge $e = \{u, v\}$ in a current tree T, one can identify locally the fundamental cycle C formed by e and the path from u to v in T. We use the *informative-labeling scheme* for nearest common ancestor (NCA) described in [6]. As for a proof-labeling scheme, every node v is provided with a *label* $\lambda(v)$. In NCA-labeling scheme, given the labels $\lambda(u)$ and $\lambda(v)$ of two nodes u and v, one can compute the label $\lambda(w) = \text{nca}(\lambda(u), \lambda(v))$ of the NCA of u and v.

We observe that, using any NCA-labeling scheme, given $\lambda(u)$, $\lambda(v)$, and the label $\lambda(w)$ of the NCA of u and v, every node can detect whether it belongs to the cycle C or not as follows: $x \in C$ if and only if

$$\begin{array}{lll} & \text{nca}(\lambda(x), \lambda(u)) = \lambda(x) & \text{and} \quad \text{nca}(\lambda(x), \lambda(v)) = \lambda(w) \\ \text{or} & \text{nca}(\lambda(x), \lambda(u)) = \lambda(w) & \text{and} \quad \text{nca}(\lambda(x), \lambda(v)) = \lambda(x) \end{array}$$

Coming up with a self-stabilizing construction of the labels in the NCA-labeling scheme of [6] essentially follows from its construction. Actually, [6] already proposed a distributed algorithm for computing the labels. Nevertheless, this is not sufficient as, for our algorithm to be silent, we also need to certify the labels. For this purpose, we show how to construct a proof-labeling scheme of the NCA-labeling scheme of [6].

Lemma 5.1: There is a proof-labeling scheme for the NCA-labeling of [6] with $O(\log n)$-bit labels. The construction of the NCA-labeling and its proof can be done in $O(n)$ rounds.

VI. Application: MST construction

Recall that Borůvska's sequential algorithm [55] (see also [34] for a distributed implementation of that algorithm, and, e.g., [19] for a most recent MST algorithm inspired by Borůvska's algorithm) proceeds in merging so-called *fragments* (a fragment is sub-tree covering all or part of the nodes). Initially, each node alone is forming a fragment. While there are more than one fragment, the algorithm scans all fragments, and, for each of them, selects the lightest outgoing edge of the fragment, adds it to the current solution formed by all selected edges so far, and merges the two fragments connected by that edge. At each iteration of the scanning phase, the number of fragments is at least halved. Therefore there are at most $\lceil \log_2 n \rceil$ such phases until one fragment remains, consisting of the desired MST.

We now describe how to define a cyclical-decreasing function ϕ fitting with the construction of MSTs. We assume given a spanning tree T, and we aim at assigning labels to the nodes, based on which we shall define the function ϕ. The labeling follows the guidelines of the proof-labeling schemes (p, v) for MST in [50], [52]. We refer to Fig. 2 for a pictural representation of the construction of our labeling.

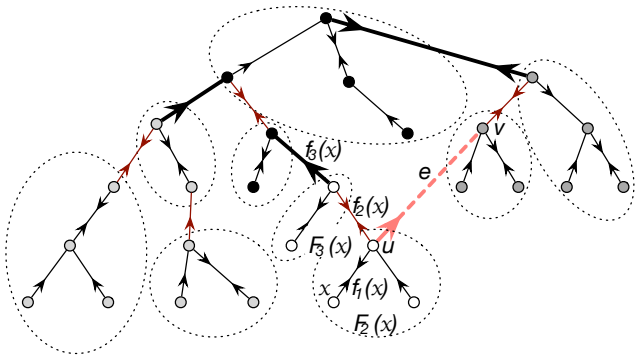

Fig. 2. *Construction of the fragments in a given spanning tree T.*

In essence, each node stores the trace of a virtual execution of Borůvska's algorithm on T, that is, every node stores the sequence of fragments it would belong to if one would execute Borůvska's algorithm on T, together with the lightest outgoing tree edge of each fragment. More precisely, each node x is given a label $\lambda(x) = (\lambda_1(x), \ldots, \lambda_k(x))$ with

$k \leq \lceil \log_2 n \rceil$. For $i = 1, \ldots, k$, $\lambda_i(x) = (F_i(x), f_i(x))$, where $F_i(x)$ is the level-i fragment including x (identified by the smallest identity of the nodes in that fragment), and $f_i(x) = (\text{ID}(a), \text{ID}(b), w(a,b))$ is a tree edge $\{a,b\}$ outgoing the fragment $F_i(x)$, together with its weight. This goes on until there is a unique level-k fragment, consisting of the whole tree T.

More specifically, for every node x, $F_1(x) = \{x\}$, and $f_1(x)$ is the lightest tree edge incident to x. On Fig. 2, these edges are depicted as the thin directed edges incident to all nodes. The collection of all these edges form a sub-forest of T, whose each tree is depicted on Fig. 2 as surrounded by dotted lines. Each tree of this sub-forest forms a level-2 fragments, and, for every node x, the edge $f_2(x)$ is the lightest outgoing tree edge of the fragment $F_2(x)$. Again, one merges level-2 fragments along these edges, to form the level-3 fragment. On Fig. 2, the level-3 fragments are the four sets of nodes with identical color (white, light grey, dark grey, and black). An so on, and so forth. On Fig. 2, since adding the lightest tree edges connecting the level-3 fragments (depicted as the three thick edges) results in the unique fragment T, we have $k = 4$, with $F_4(x) = T$, and $f_4(x) = \perp$ for every node x.

We denote by k the number of levels in the above construction of fragments in T. We define $\phi(T) = kn - \sum_{x \in V} \phi_x(T)$ where, for every node x, $\phi_x(T)$ denotes the largest index i, $0 \leq i \leq k$, such that, for every $j = 1, \ldots, i$, $f_j(x)$ is the minimum-weight outgoing edge of fragment $F_j(x)$ in G. Note that if T is a MST in G, then $\phi(T) = 0$ since, for every node x, we have $\phi_x(T) = k$. On the other hand, if T is not a MST in G, then some of the tree edges are not the minimum-weight outgoing edges of their fragments in G, and hence $\phi_x(T) < k$ for at least one node x, resulting in $\phi(T) > 0$. Therefore, we do have: T is a MST $\iff \phi(T) = 0$.

Let us now focus on a spanning tree T such that $\phi(T) > 0$, and a node x with $\phi_x(T) = i < k$. It means that the edge $f_{i+1}(x)$ is not the minimum weight outgoing edge of the fragment $F_{i+1}(x)$. On Fig. 2, the fragment $F_3(x)$ is the set of white nodes, but the outgoing tree edge $f_3(x) \in T$ of fragment $F_3(x)$ is not the outgoing edge of minimum weight of $F_3(u)$, which, in this case, is edge $e = \{u, v\}$. Tarjan's red rule specifies that the edge $f \in T$ of maximum weight along the fundamental cycle of $T + e$ cannot be any MST of G. Replacing f by e in the tree, i.e., replacing T by $T + e - f$ increases the fit between the label and the tree, in the sense that $\phi(T + e - f) < \phi(T)$. Therefore, ϕ is cyclical-decreasing, and Algorithm 2 below is a instantiation of Algorithm 1 for MST.

In this algorithm, one is using labels of size $O(\log^2 n)$ bits, which is optimal [50], and the construction of these labels in a silent self-stabilizing manner can be done using standard *convergecast* operations gathering information at the root of the current tree, and *broadcast* operations diffusing information from that root, in poly(n) rounds. Finding the appropriate e and f to be switched just need to look for the heaviest edge f along the fundamental cycle $T + e$ where e is the lightest outgoing edge of a fragment. Again, standard convergecast and broadcast operations enable to compute these edges in poly(n) rounds, using $O(\log n)$ bits of memory. Finally, $\phi_{max} \leq n\lceil \log n \rceil$. Therefore, the family MST is efficiently locally searchable, and, by Theorem 3.1, we get:

Algorithm 2 *A PLS-guided version of Borůvska's Algorithm*

Require: a connected graph $G = (V, E)$
construct a spanning tree T of G
for all nodes u **do**
  compute $F_i(u)$ and $f_i(u)$, $i = 1, \ldots, k$
end for
while $\phi(T) \neq 0$ **do**
  let u and i such that $\phi_u(T) = i < k$
  let $e =$ min-weight outgoing edge of $F_{i+1}(u)$
  let $f =$ max-weight edge on $T + e$
  $T \leftarrow T + e - f$
  for all nodes u **do**
    update $F_i(u)$ and $f_i(u)$, $i = 1, \ldots, k$
  end for
end while
output T

Corollary 6.1: There is a silent self-stabilizing implementation of Algorithm 2 constructing a MST in any graph G, in poly(n) *rounds, and optimal space complexity* $O(\log^2 n)$ *bits of memory at each node.*

VII. Beyond unique edge-replacements

For certain constrained spanning tree construction tasks, it may not always be possible to improve the current solution by replacing one edge by another, like in Algorithm 1. This is typically the case of constructing a minimum-degree spanning tree (MDST), as it will appear clear in the Section VIII. On the other hand, improvement is always possible if one if allowed to perform an arbitrary sequence of switches resulting in $T' = T \cup \{e_1, \ldots, e_k\} \setminus \{f_1, \ldots, f_k\}$, where $e_i \notin T$ and $f_i \in T$, for $i = 1, \ldots, k$. We say that an ordered sequence of edge-pairs (e_i, f_i), $i = 1, \ldots, k$ is *well nested* if, for every i, we have: (a) $e_i \notin T$, (b) f_i is an edge of the fundamental cycle of $T + e_i$, and (c) for every $j > i$, the edges e_j and f_j are connecting nodes in the same subtree of T in the forest resulting from T by removing the edges of all fundamental cycles in $T + e_1, \ldots, T + e_i$. We can then generalize the notion of cyclical-decreasing potential function defined in Section III by introducing *nest-decreasing* potential functions, defined by replacing the existence of a pair of edges $e \notin T$ and f in the fundamental cycle of $T + e$, by the existence of a well nested sequence of edges pairs (e_i, f_i), $i = 1, \ldots, k$. If the existence of such a potential function ϕ is secured for a family $\mathcal{F}$ of spanning trees, then one can apply again a local-search greedy algorithm, such as Algorithm 3 below.

Lemma 3.1 then generalizes to the following, replacing the time and space to find an appropriate fundamental cycle by the time and space to find an appropriate well nested sequence.

Lemma 7.1: If a family $\mathcal{F}$ of spanning trees admits a nest-decreasing potential function ϕ, then there is a silent self-stabilizing implementation of Algorithm 3 solving task $\mathcal{F}$ with time complexity $\phi_{max} \cdot (t_{label} + t_{find\text{-}nest} + O(n))$ rounds, and with registers of size $s_{label} + s_{find\text{-}nest} + O(\log n)$ bits.

Algorithm 3 *PLS-guided spanning tree construction II*

Require: a graph G, and a proof-labeling scheme (p, v) for a family $\mathcal{F}$ of spanning trees
construct a spanning tree T of G
while $\phi(T) \neq 0$ **do**
  find a well nested seq. of edges e_i, f_i, $i = 1, \ldots, k$, s.t. $\phi(T \cup \{e_1, \ldots, e_k\} \setminus \{f_1, \ldots, f_k\}) < \phi(T)$
  $T \leftarrow T \cup \{e_1, \ldots, e_k\} \setminus \{f_1, \ldots, f_k\}$
end while
output T

Similarly, one can extend the notion of efficiently locally searchable families $\mathcal{F}$, replacing the conditions on finding the appropriate fundamental cycle by the same conditions but on finding the appropriate well nested sequence. This enables to derive the following:

Theorem 7.1: For every efficiently locally searchable family $\mathcal{F}$ of spanning trees, there is a silent self-stabilizing implementation of Algorithm 3 solving task $\mathcal{F}$ in $\text{poly}(n)$ *rounds, with registers of size* $\text{OPT} + O(\log n)$ *bits.*

VIII. Application: MDST construction

Unfortunately, the situation for near-MDST (i.e., $+1$ additive approximation for MDST) turns out to be more complex than for MST. Indeed, it is unlikely that there is a proof-labeling scheme for near-MDST using labels of logarithmic size.

Proposition 8.1: Unless NP $=$ co-NP*, there are no proof-labeling schemes for near-*MDST *involving* $O(\text{poly}(n))$ *computation time at each node of n-node graphs. Thus, in particular, unless* NP $=$ co-NP*, there are no proof-labeling schemes for near-*MDST *using labels of size* $O(\log n)$ *bits.*

A direct consequence of Proposition 8.1 is that it is unlikely that there exists a silent, time-efficient self-stabilizing algorithm constructing and stabilizing on spanning trees with degrees at most $\text{OPT} + 1$. Therefore, we shall now focus on constructing trees belonging to a subclass of spanning trees with degrees at most $\text{OPT} + 1$. We define FR-trees, named after Fürer and Raghavachari.

Definition 8.1: T is a FR-tree *in a graph G if T is a degree-k spanning tree of G whose every node can be marked "good" or "bad" such that the following three properties hold: (1) every node with degree k in T is marked bad, (2) every node with degree at most $k-2$ in T is marked good, and (3) there are no edges in G between two good nodes in two different trees of the forest resulting from T by removing the bad nodes (and their incident edges).*

Removing all bad nodes from a FR-tree T results in a forest whose trees are called *fragments*. Fürer and Raghavachari [33] have proved that every (connected) graph has a spanning tree that is a FR-tree (e.g., an Hamiltonian path in an Hamiltonian graph is a FR-tree by marking all nodes bad). Theorem 2.2 in [33] states that the degree k of any FR-tree satisfies $k \leq \text{OPT} + 1$. However, not all spanning trees of degree OPT or $\text{OPT} + 1$ are FR-trees. The algorithm of Fürer and Raghavachari, of which we shall provide a distributed self-stabilizing implementation, produces a FR-tree. In order to keep the algorithm silent, we use a proof-labeling scheme for FR-trees, directly inspired from Definition 8.1.

Lemma 8.1: There is a proof-labeling scheme for FR-trees with $O(\log n)$-bit labels. Any silent self-stabilizing FR-tree construction algorithm requires $\Omega(\log n)$-bit memory.

The algorithm by Fürer and Raghavachari [33] is presented in Algorithm 4 below. It was proved in [33] that this algorithm produces a FR-tree (and therefore a tree with degree at most $\text{OPT} + 1$), in time $O(mn \log(n)\alpha(m, n))$ in n-node m-edge graphs, where α is the inverse Ackerman function. Interestingly enough, this algorithm has precisely the structure of the PLS-guided spanning tree construction of Algorithm 3. Moreover, the sequence of edges (e_i, f_i) as defined in the algorithm are well nested.

Algorithm 4 *Fürer and Raghavachari Algorithm [33]*

Require: a connected graph $G = (V, E)$
construct a spanning tree T of G
repeat
  let d be the degree of T
  mark vertices of degree d and $d-1$ as bad, and all other vertices as good
  let F be the set of fragments resulting from removing bad nodes from T
  while $\exists e \in E$ between two different fragments and all degree-d vertices are bad **do**
    mark good all vertices in the cycle C in $T \cup \{e\}$
    update set of fragments F
  end while
  if some degree-d vertex is good **then**
    let w be a vertex of degree d marked as good
    Let e_i, f_i, $i = 1, \ldots, k$ be a sequence of improve--ments enabling to decrease $\deg(w)$
    $T \leftarrow T \cup \{e_1, \ldots, e_k\} \setminus \{f_1, \ldots, f_k\}$
  end if
until all degree-d vertices are bad
output T

Let $\phi(T) = (n\Delta_T + N_T)(1 - \mathbf{1}_{\text{FR}}(T))$ where Δ_T denotes the maximum degree of T, N_T denotes the number of nodes with degree Δ_T, and $\mathbf{1}_{\text{FR}}(T))$ denotes the indicator function of FR-trees. The proof-labeling scheme of Lemma 8.1 enables to detect whether $\mathbf{1}_{\text{FR}}(T) = 0$ or 1. We get that all the conditions of Theorem 7.1 are satisfied, and thus we derive the following [14]:

Corollary 8.1: There exist a silent self-stabilizing construction algorithm for MDST, stabilizing on a class of spanning trees with degree at most $\text{OPT}+1$*, using optimal-size $O(\log n)$-bits registers, converging in* $\text{poly}(n)$ *rounds, and performing polynomial-time computations at each node.*

REFERENCES

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

[2] Y. Afek and S. Dolev. Local Stabilizer. *J. Parallel Distrib. Comput.* **62**(5): 745-765 (2002)

[3] Y. Afek, S. Kutten, and M. Yung. Memory-efficient self stabilizing protocols for general networks. In proc. WDAG 1990.

[4] Y. Afek, S. Kutten, and M. Yung. The local detection paradigm and its applications to self-stabilization. *TCS* **186**(1-2):199-229 (1997)

[5] T. Akiyama, T. Nishizeki, and N. Saito. NP-completeness of the Hamiltonian cycle problem for bipartite graphs. *J. of Information Processing* **3**(2): 7376 (1980)

[6] S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe. Nearest Common Ancestors: A Survey and a New Algorithm for a Distributed Environment. *Theory of Comput. Systems* **37**:441-456 (2004)

[7] A. Arora and M. Gouda. Distributed reset. *IEEE Trans. Computers* **43**(9):1026-1038 (1994)

[8] B. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir, and G. Varghese. Time optimal self- stabilizing synchronization. In proc. STOC 1993.

[9] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction. In proc. FOCS 1991.

[10] J. Beauquier, S. Delaët, S. Dolev, and S. Tixeuil. Transient fault detectors. *Distributed Computing* **20**(1): 39-51 (2007)

[11] J. Beauquier, M. Gradinariu, and C. Johnen. Memory Space Requirements for Self-Stabilizing Leader Election Protocols. In proc. PODC'99.

[12] L. Blin, F. Boubekeur, S. Dubois. A Self-Stabilizing Memory Efficient Algorithm for the Min-Diameter Spanning Tree. In proc. IPDPS 2015

[13] L. Blin, S. Dolev, M. Gradinariu, and S. Rovedakis. Fast self-stabilizing minimum spanning tree construction. In proc. DISC 2010.

[14] L. Blin, and P. Fraigniaud. Polynomial-Time Space-Optimal Silent Self-Stabilizing Minimum-Degree Spanning Tree Construction. arXiv 2014

[15] L. Blin, P. Fraigniaud, and B. Patt-Shamir. On Proof-Labeling Schemes versus Silent Self-stabilizing Algorithms. In proc. SSS 2014.

[16] L. Blin, M. Gradinariu, and S. Rovedakis. Self-stabilizing minimum degree spanning tree within one from the optimal degree. *J. Parallel Distrib. Comput.* **71**(3):438–449 (2011)

[17] L. Blin, M. Gradinariu, S. Rovedakis, and S. Tixeuil. A new self-stabilizing minimum spanning tree construction with loop-free property. In proc. DISC 2009.

[18] J. Burman and S. Kutten. Time optimal asynchronous self-stabilizing spanning tree. In proc. DISC 2007.

[19] B. Chazelle. A minimum spanning tree algorithm with Inverse-Ackermann type complexity. *J. ACM* **47**(6): 1028-1047 (2000)

[20] N.-S. Chen, H.-P. Yu, and S.-T. Huang. A Self-Stabilizing Algorithm for Constructing Spanning Trees. *Inf. Proc. Letters* **39**(3): 147-151 (1991)

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

[22] A. Cournier. A new polynomial silent stabilizing spanning-tree construction algorithm. In proc. SIROCCO 2009.

[23] A. Cournier, S. Devismes, and V. Villain. Light enabling snap-stabilization of fundamental protocols. *ACM Transactions on Autonomous and Adaptive Systems* **4**(1) (2009)

[24] A. Cournier, S. Rovedakis, and V. Villain. The first fully polynomial stabilizing algorithm for bfs tree construction. In proc. OPODIS 2011.

[25] A. Datta, L. Larmore, and P. Vemula Self-stabilizing leader election in optimal space under an arbitrary scheduler. *Theor. Comput. Sci.* **412**(40):5541-5561 (2011)

[26] E. Dijkstra. Self-stabilizing Systems in Spite of Distributed Control. *Commun. ACM* **17**(11): 643-644 (1974)

[27] S. Dolev. Self-Stabilization. *MIT Press* (2000)

[28] S. Dolev, M. G. Gouda, and M. Schneider. Memory Requirements for Silent Stabilization. *Acta Inf.* **36**(6):447-462 (1999)

[29] S. Dolev and T. Herman, Superstabilizing protocols for dynamic distributed systems. *Chicago J. of Theo. Comp. Sc.* **1997**:140 (1997)

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

[31] P. Fraigniaud. Approximation Algorithms for Minimum-Time Broadcast under the Vertex-Disjoint Paths Mode. In proc. ESA 2001.

[32] P. Fraigniaud, A. Korman, and E. Lebhar. Local MST computation with short advice. In proc. SPAA 2007.

[33] M. Fürer and B. Raghavachari. Approximating the Minimum-Degree Steiner Tree to within One of Optimal. *J. Alg.* **17**(3): 409-423 (1994)

[34] R. Gallager, P. Humblet, P. Spira. A Distributed Algorithm for Minimum-Weight Spanning Trees. *ACM Trans. Program. Lang. Syst.* **5**(1): 66-77 (1983)

[35] M.R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete problems. In proc. STOC 1974.

[36] B. Gavish. Topological design of centralized computer networks: Formulation and algorithms. *Networks* **12**:355-377 (1982)

[37] E. Gilbert and E. Moore. Variable-length binary encodings. *Bell System Technical Journal* **38**:933–967, 1959.

[38] S. Gupta, A. Bouabdallah, P. Srimani. Self-stabilizing protocol for shortest path tree for multicast routing in mobile networks. In EuroPar 2000.

[39] S. Gupta and P. Srimani. Self-stabilizing multicast protocols for ad hoc networks. *J. Parallel Distrib. Comput.* **63**(1):87–96 (2003)

[40] D. Harel, and R. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal Computing* **13**(2), 338–355 (1984)

[41] L. Higham and Z. Liang. Self-stabilizing minimum spanning tree construction on message-passing networks. In proc. DISC 2001.

[42] S.-T. Huang and N.-S. Chen. A self-stabilizing algorithm for constructing breadth-first trees. *Inf. Process. Lett.* **41**(2):109–117 (1992)

[43] S.-T. Huang and N.-S. Chen. Self-stabilizing depth-first token circulation on networks. *Distributed Computing* **7**(1):61–66 (1993)

[44] S.-T. Huang. A self-stabilizing algorithm for the shortest path problem assuming read/write atomicity. *J. Comput. Syst. Sci.*, 71(1):70–85, 2005.

[45] R. Kawatra. A multiperiod degree constrained minimal spanning tree problem. *European Journal of Operational Research* **143**:53-63 (2002)

[46] A. Itai, C. H. Papadimitriou, and J. L. Szwarcfiter. Hamiltonian paths in grid graphs. *SIAM J. Comput.* **11**(4): 676-686 (1982)

[47] G. Itkis, L. A. Levin. Fast and Lean Self-Stabilizing Asynchronous Protocols. In proc. FOCS 1994.

[48] C. Johnen. Memory-efficient self-stabilizing algorithm to construct BFS spanning trees. In proc. SSS 1997.

[49] P. Klein, R. Krishnan, B. Raghavachari, R. Ravi. Approximation algorithms for finding low-degree subgraphs. *Networks* **44**: 203-215, 2004

[50] A. Korman, S. Kutten. Distributed verification of minimum spanning tree. *Distributed Computing* **20**: 253-266 (2007)

[51] A. Korman, S. Kutten, T. Masuzawa. Fast and compact self stabilizing verification, computation, and fault detection of MST. In PODC 2011.

[52] A. Korman, S. Kutten, and D. Peleg. Proof labeling schemes. *Distributed Computing* **22**(4): 215-233 (2010)

[53] A. Kosowski and L. Kuszner. A self-stabilizing algorithm for finding a spanning tree in a polynomial number of moves. In proc. PPAM 2005.

[54] S.C. Narula and C.A. Ho. Degree-constrained minimum spanning tree. *Computors and Operational Research* **7**:239-249 (1980)

[55] J. Nesetril, E. Milková, and H. Nesetrilová. Otakar Boruvka on minimum spanning tree problem. *Disc. Math.* **233**(1-3): 3-36 (2001)

[56] David Peleg. Distributed Computing: A Locality-Sensitive Approach. *SIAM* (2000).

[57] David Peleg. Informative labeling schemes for graphs. *Theor. Comput. Sci.* **340**(3):577-593 (2005)

[58] M. Singh, L. C. Lau. Approximating minimum bounded degree spanning trees to within one of optimal. In proc.STOC 2007.

[59] The IRIS project: *http://www.anr-iris.fr.*