The First Fully Polynomial Stabilizing Algorithm for BFS Tree Construction[⋆]

Alain Cournier[1], Stéphane Rovedakis[2], and Vincent Villain[1]

[1] Laboratoire MIS, Université de Picardie, 33 Rue St Leu, 80039 Amiens Cedex 1, France
{alain.cournier,vincent.villain}@u-picardie.fr
[2] Laboratoire CEDRIC, CNAM, 292 Rue St Martin, 75141 Paris Cedex 03, France
stephane.rovedakis@cnam.fr

Abstract. The construction of a spanning tree is a fundamental task in distributed systems which allows to resolve other tasks (i.e., routing, mutual exclusion, network reset). In this paper, we are interested in the problem of constructing a *Breadth First Search* (BFS) tree. *Stabilization* is a versatile technique which ensures that the system recover a correct behavior from an arbitrary global state resulting from transient faults.

A *fully polynomial* algorithm has a round complexity in $O(d^a)$ and a step complexity in $O(n^b)$ where d and n are the diameter and the number of nodes of the network and a and b are constants. We present the first fully polynomial stabilizing algorithm constructing a BFS tree under a distributed daemon. Moreover, as far as we know, it is also the first fully polynomial stabilizing algorithm for spanning tree construction. Its round complexity is in $O(d^2)$ and its step complexity is in $O(n^6)$.

To our knowledge, since in general the diameter of a network is much smaller than the number of nodes ($\log(n)$ in average instead of n), this algorithm reaches the best compromise of the literature between the complexities in terms of rounds and in terms of steps.

Keywords: Distributed systems, Fault-tolerance, Stabilization, Spanning tree construction.

1 Introduction

The construction of spanning trees is a fundamental problem in the field of distributed systems. A spanning tree is a virtual structure which contains no cycle and interconnects all the nodes of a network. In distributed systems, the construction of a spanning tree is commonly used to design algorithms resolving other distributed tasks, like routing, token circulation or message broadcasting in a network. Spanning trees are also used to obtain algorithms resolving a particular distributed problem with a better time complexity compared to algorithms for the same problem which do not use this structure. There are many different spanning tree construction problems guaranteeing various properties, e.g., the construction of a depth first search (DFS) tree, a spanning tree of minimum weight or a spanning tree of minimum diameter. A crucial class of

[⋆] This work has been supported in part by the ANR project SPADES (08-ANR-SEGI-025).

spanning trees is the construction of a Breadth First Search (BFS) tree, which contains shortest paths (in hops) from every node to the root of the tree. This structure is mainly used in networks to quickly broadcast information from a source node. When a cost is associated to communication links, this problem is known as the construction of a Shortest Path tree.

Self-stabilization introduced first by Dijkstra in [16] and later publicized by several books [17,23] is one of the most versatile techniques to handle transient faults arising in distributed systems. A distributed algorithm is self-stabilizing if starting from any arbitrary global state (due to faults or attacks) the system is able to recover from this catastrophic situation in finite time without external (e.g., human) intervention. As self-stabilization makes no hypothesis about the nature or the extent of the faults, this paradigm can also be used to handle dynamic changes on the network topology since these modifications are seen as faults by the system. Another kind of stabilization was introduced by Bui *et al* [4], called *snap-stabilization*. These algorithms have the ability to always guarantee a correct system behavior according to the specifications of the problem to be solved, starting from any arbitrary global state.

Related work. Due to the importance of the construction of spanning trees, there are a lot of works which study this task. A survey on several self-stabilizing tree constructions can be found in [19]. Moreover, Table 1 summarizes the time complexities (round and step complexities) of some self-stabilizing tree construction algorithms. The number of steps required to compute a solution is an important criterion since it reflects the number of messages exchanged by an algorithm; especially for a self-stabilizing algorithm for which each node has to send messages to its neighbors in order to inform them that its state has been changed. However, few works give a step complexity analysis as can be seen in Table 1, except for [22,11,10,13] and [9] which presented an algorithm improving the step complexity to $\Theta(n^2)$ steps for the construction of an arbitrary spanning tree (with n the number of nodes of the network). Another essential criterion concerns the round complexity of a distributed algorithm, that is to have a round complexity only function of the network diameter (which is much smaller than the size of network for most network topologies). Some of the algorithms cited in Table 1 are optimal in terms of rounds for the construction of an arbitrary spanning tree or a BFS tree.

From the above discussion, we give a characterization for self-stabilizing distributed algorithms having an efficient complexity to solve a task, called fully polynomial algorithms. A *fully polynomial* algorithm has a round complexity in $O(d^a)$ and a step complexity in $O(n^b)$ where d and n are the diameter and the number of nodes of the network and a and b are constants. As presented in Table 1, the existing self-stabilizing spanning tree construction algorithms with a polynomial step complexity requires $\Omega(n)$ rounds, or a round complexity of $\Theta(\max(d^2, n))$ for the construction of a BFS tree. To our knowledge, no fully polynomial stabilizing algorithm was given for the construction of a spanning tree. Therefore, a legitimate question can be the following: Is it possible to construct in a self-stabilizing manner a spanning tree with a polynomial step complexity and a round complexity lower than $\Theta(\max(d^2, n))$?

Contributions. In this paper, we present the first fully polynomial stabilizing algorithm for the construction of a spanning tree with a round complexity lower than $\Theta(\max(d^2, n))$. Notice that the algorithm presented in [13] does not satisfy the

Table 1. Distributed stabilizing algorithms for the construction of spanning trees. n, d and Δ are respectively the number of nodes, the diameter and the maximum degree in the network, while N is an upper bound of n and Max is the maximum height value in the tree of a node in the initial configuration. The *silent* property for a self-stabilizing algorithm is to guarantee that when a legitimate configuration is reached the values stored in the registers do not change anymore.

	References	Round complexity	Step complexity	Memory complexity	Silent property
BFS	[2]	$O(N^2)$	Undetermined	$O(\log(n))$	Yes
	[18]	$O(d)$	Undetermined	$O(\Delta \log(n))$	Yes
	[1]	$O(n^2)$	Undetermined	$O(\log(n))$	Yes
	[20]	$\Theta(d)$	$O(n(Max+d)^n)^4$	$O(\log(n))$	Yes
	[3]	$O(d)$	Undetermined	$O(\log^2(n))$	Yes
	[21]	$\Omega(d^2)$	Undetermined	$O(\log(\Delta))$	No
	[5]	$O(d)$	Undetermined	$O(\log^2(n))$	Yes
	[15]	$O(n)$	Undetermined	$O(\log(n))$	Yes
	[13]	$\Theta(d^2+n)$	$O(\Delta n^3)$	$O(\log(n))$	No
	This paper	$O(d^2)$	$O(n^6)$	$O(\log(n))$	Yes
Any	[6]	$O(n)$	$\Omega(2^n)^4$	$O(\log(n))$	Yes
	[22]	$O(n)$	$\Theta(n^2 d)$	$O(\log(n))$	Yes
	[9]	$\Theta(n)$	$\Theta(n^2)$	$O(\log(n))$	Yes
DFS	[7]	$O(dn\Delta)$	Undetermined	$O(n\log(\Delta))$	Yes
	[11]	$O(n^2)$	$O(n^3)$	$O(\log(n))$	Yes
	[10]	$O(n)$	$O(n^2)$	$O(n\log(n))$	Yes
	[13]	$O(n)$	$O(\Delta n^3)$	$O(\log(\Delta+n))$	No

definition of a fully polynomial algorithm since it has a round complexity which is related with the network size. Our algorithm computes a BFS tree in $O(d^2)$ rounds with a polynomial number of steps in $O(n^6)$ (the step complexity is $O(mn^4)$ and $m << n^2$) under a distributed daemon without any fairness assumptions, with d the diameter, m the number of edges and n the number of nodes in the network. To our knowledge, since in general the diameter of a network is much smaller than the number of nodes ($\log(n)$ in average instead of n), this algorithm reaches the best compromise of the literature between the complexities in terms of rounds and in terms of steps. Moreover, this BFS tree construction is based on a snap-stabilizing algorithm given in this paper resolving the Question-Answer problem, in which each node requests a permission (delivered by a subset of network nodes) in order to perform a defined computation, which is of independent interest.

Outline of the paper. The paper is organized as follows. In Section 2 we present the model assumed in this paper. We then present a fully polynomial stabilizing algorithm to construct a BFS tree in Section 3, based on a snap-stabilizing algorithm to the Question-Answer problem given in Section 4. We describe in Section 5 how these stabilizing algorithms are composed together and give an explanation about the time complexity to solve the BFS tree problem. Finally, we conclude in the last section.[1]

[1] This is detailed in the analysis given in [8].

2 Model

Notations. We consider a network as an undirected connected graph $G = (V,E)$ where V is a set of nodes (or *processors*) and E is the set of *bidirectional asynchronous communication links*. We state that n is the size of G ($|V| = n$). We assume that the network is *rooted*, i.e., among the processors, we distinguish a particular one, *r*, which is called the *root* of the network. In the network, p and q are neighbors if and only if a communication link (p,q) exists (i.e., $(p,q) \in E$). Every processor p can distinguish all its links. To simplify the presentation, we refer to a link (p,q) of a processor p by the *label* q. We assume that the labels of p, stored in the set $Neig_p$, are locally ordered by $\prec_p$. We also assume that $Neig_p$ is a constant input from the system. Δ is the maximum degree of the network (i.e., the maximal value among the local degrees of the processors). A tree $T = (V_T, E_T)$ is an acyclic connected subgraph such that $V_T \subseteq V$ and $E_T \subseteq E$, where the root of tree T is noted by $root(T)$. Moreover, any processor has a *parent* in a tree T which is the neighbor on the path leading to $root(T)$. A processor $p \in V_T$ with at least two neighbors in tree T is called an *internal* processor and a *leaf* processor otherwise.

Programs. In our model, protocols are *semi-uniform*, i.e., each processor executes the same program except *r*. We consider the local shared memory model of computation. In this model, the program of every processor consists in a set of *variables* and an *ordered finite set of actions* inducing a *priority*. This priority follows the order of appearance of the actions into the text of the protocol. A processor can write to its own variable only, and read its own variables and that of its neighbors. Each action is constituted as follows: $< label > :: < guard > \rightarrow < statement >$. The guard of an action in the program of p is a boolean expression involving variables of p and its neighbors. The statement of an action of p updates one or more variables of p. An action can be executed only if its guard is satisfied. The *state* of a processor is defined by the value of its variables. The *state* of a system is the product of the states of all processors. We will refer to the state of a processor and the system as a (*local*) *state* and (*global*) *configuration*, respectively. We note $\mathcal{C}$ the set of all possible configuration of the system. Let $\gamma \in \mathcal{C}$ and A an action of p ($p \in V$). A is said to be *enabled* at p in γ if and only if the guard of A is satisfied by p in γ. Processor p is said to be *enabled* in γ if and only if at least one action is enabled at p in γ. When several actions are enabled simultaneously at a processor p: only the priority enabled action can be activated.

Let a distributed protocol P be a collection of binary transition relations denoted by $\mapsto$, on $\mathcal{C}$. A *computation* of a protocol P is a *maximal* sequence of configurations $e = (\gamma_0,\gamma_1,...,\gamma_i,\gamma_{i+1},...)$ such that, $\forall i \geq 0, \gamma_i \mapsto \gamma_{i+1}$ (called a *step*) if γ_{i+1} exists, else γ_i is a terminal configuration. *Maximality* means that the sequence is either finite (and no action of P is enabled in the terminal configuration) or infinite. All computations considered here are assumed to be maximal. $\mathcal{E}$ is the set of all possible computations of P.

As we already said, each execution is decomposed into steps. Each step is shared into three sequential phases atomically executed: (i) every processor evaluates its guards, (ii) a *daemon* (also called *scheduler*) chooses some enabled processors, (iii) each chosen processor executes its priority enabled action. When the three phases are done, the next step begins.

A *daemon* can be defined in terms of *fairness* and *distributivity*. In this paper, we use the notion of *unfairness*: the *unfair* daemon can forever prevent a processor from executing an action except if it is the only enabled processor. Concerning the *distributivity*, we assume that the daemon is *distributed* meaning that, at each step, if one or more processors are enabled, then the daemon chooses at least one of these processors to execute an action.

We consider that any processor p executed a *disabling action* in the computation step $\gamma_i \mapsto \gamma_{i+1}$ if p was *enabled* in γ_i and not enabled in γ_{i+1}, but did not execute any protocol action in $\gamma_i \mapsto \gamma_{i+1}$. The disabling action represents the following situation: at least one neighbor of p changes its state in $\gamma_i \mapsto \gamma_{i+1}$, and this change effectively made the guard of all actions of p false in γ_{i+1}.

To compute the time complexity, we use the definition of *round*. This definition captures the execution rate of the slowest processor in any computation. Given a computation e ($e \in \mathcal{E}$), the *first round* of e (let us call it e') is the minimal prefix of e containing the execution of one action (an action of the protocol or a disabling action) of every enabled processor from the initial configuration. Let e'' be the suffix of e such that $e = e'e''$. The *second round* of e is the first round of e'', and so on.

3 Spanning Tree Construction

In this section, we are interested in the problem of constructing a tree spanning all the processors of the network. We consider there is a particular *root* processor, noted r, which is used to construct a spanning tree. More precisely, we consider the construction of a *Breadth First Search* (BFS) tree rooted at processor r. We can define a BFS tree as in Definition 1.

Definition 1 (BFS Tree). *Let $G = (V, E)$ be a network and r a node called the* root. *A graph $T = (V_T, E_T)$ of G is called a* Breadth First Search *tree if the following conditions are satisfied:*

1. *$V_T = V$ and $E_T \subseteq E$, and*
2. *T is a connected graph (i.e., there exists a path in T between any pair of nodes $x, y \in V_T$) and $|E_T| = |V| - 1$, and*
3. *For each node $p \in V_T$, there exists no shorter path (in hops) between p and r in G than the path between p and r in T.*

We give a formal specification to the problem of constructing a BFS tree, stated in Specification 1.

Specification 1 (Tree Construction). *Let $\mathcal{C}$ the set of all possible configurations of the system. An algorithm $\mathcal{A}_{\mathcal{BFS}}$ solving the problem of constructing a stabilizing BFS tree satisfies the following conditions:*

[TC1] *Algorithm $\mathcal{A}_{\mathcal{BFS}}$ reaches a set of terminal configurations $\mathcal{T} \subseteq \mathcal{C}$ in finite time, and*

[TC2] *Every configuration $\gamma \in \mathcal{T}$ satisfies Definition 1.*

3.1 Breadth First Search Tree Algorithm

In this section, we present a snap-stabilizing algorithm, called $\mathcal{BFS}$, to construct a BFS tree. Algorithm $\mathcal{BFS}$ is a semi-uniform algorithm, this means that exactly one of the processors, called the *root* and denoted r, is distinguished. This distinguished processor is used in Algorithm $\mathcal{BFS}$ as the root of the spanning tree.

Algorithm $\mathcal{BFS}$ is a composition of two algorithms. Algorithm 1 is based on the fact that a processor has to choose a neighbor with the minimal distance to the root as its parent in the tree. It is well known that this common idea is enough to get a round complexity in $O(d)$, but does not ensure a step complexity in $O(n^b)$[2] So we allow a processor to connect to a neighbor only if this neighbor is in the tree rooted at r and in the shortest path to r. The detection of such neighbors is assigned to Algorithm 2 (see Section 4) which can be seen as an oracle by Algorithm 1. The second role of Algorithm 1 is to remove the abnormal trees, i.e., those that are not rooted at r.

Variables. We define below the different variables used by Algorithm 1. For Algorithm 1, we characterize r by the predicate $Allowed$ (i.e., $Allowed(p) \equiv (p = r)$, $\forall p \in V$).

Shared variable. Each processor $p \in V$ has a local shared variable $p.Req$ which is used by Algorithm 1 to monitor Algorithm 2 at p. This shared variable can take four values: ASK, $WAIT$, REP, and OUT. By setting the shared variable $p.Req$ to ASK, Algorithm 1 informs Algorithm 2 that a permission from the root of the tree that p belongs to is needed at p. In this case, Algorithm 2 tries to send a request and to obtain a permission for p if it is possible (i.e., if p belongs to an allowed tree and this request has the highest priority during enough time). If a permission is delivered to processor p, then Algorithm 2 sets this shared variable to REP in order to inform Algorithm 1. Then, every neighbor of p can execute Algorithm 1 to join the tree that p belongs to. When there is no neighbor of p to connect, then Algorithm 1 sets $p.Req$ to OUT which allows to Algorithm 1 to request another permission through Algorithm 2 if needed.

Local variables. Each processor $p \in V$ maintains three local variables:

- $p.P$: it gives the parent of p in the tree it belongs to, $p.P = \perp$ for processor $p = r$.
- $p.L$: it stores the level (or height) of p in the tree it belongs to, $p.L = 0$ for processor $p = r$.
- $p.S$: it defines the status of processor p. It can take two values: E if p does not belong to a tree rooted to a processor x satisfying Predicate $Allowed(x)$, C otherwise. We have $p.S = C$ for processor $p = r$.

Algorithm Description. As described before, we consider a forest $\mathcal{F}$ of trees and a distinguished processor r which is the only processor authorized to deliver permissions in the network (i.e., $Allowed(p) \equiv (p = r)$ for every processor $p \in V$). We can notice

[2] Indeed, this approach is used in [20] to construct a BFS tree with a round complexity in $\Theta(d)$ but with a step complexity in $\Omega(Max \times n^2)$, as demonstrated in [8]. However, Max is an upper bound of n and can be arbitrary high with respect to n so the step complexity can be at least exponential. Note that the gap between the lower and the upper bound (see Table 1) of the step complexity lead us to think that the lower bound in [8] is not tight.

that in a tree there is a strong constraint between the level of a processor and the level of its parent in the tree: For any processor $p \neq r$, the level of p's parent must be equal to p's level minus 1. Therefore, the root of a tree in forest $\mathcal{F}$ is either (i) processor r, or (ii) a processor $p \neq r$ such that $p.L \leq (p.P).L$ (it is used to detect cycles in the network). Since we want to construct a spanning tree, in case (ii) we say that processor p is an *abnormal root*. Moreover, any processor $p \neq r$ in a tree in $\mathcal{F}$ rooted at an abnormal root belongs to an *abnormal tree*. Every processor $p \in V$ in an abnormal tree can execute E-$action$ to change its Status to E (i.e., $p.S = E$) and to inform its descendants in the tree (see the formal description of Algorithm 1). Note that to reduce the number of moves executed by Algorithm $\mathcal{BFS}$, a processor $p \in V$ in an abnormal tree does not ask any permission. Processor p waits until a neighbor q in the tree rooted at r authorizes p to connect to q.

```
Algorithm 1.   Spanning Tree Construction for any $p \in V$

Inputs: $Neig_p$: set of (locally) ordered neighbors of $p$;
Shared variable: $p.Req \in \{ASK, WAIT, REP, OUT\}$;
..........................................................................
Macros:
 $Child(p)$       $= \{q \in Neig_p :: q.P = p \wedge q.L = p.L + 1\}$
 $Parent(p)$      $= p.P$
 $Height(p)$      $= p.L$
 $ChPar(p)$       $= \{q \in Neig_p \backslash Child(p) :: q.S = C\}$
 $MinChPar(p)$    $= \min\{q \in ChPar(p) :: \forall t \in ChPar(p), q.L \leq t.L\}$
..........................................................................
Global Predicates:
 $GoodT(p)$       $\equiv p.S \neq E \wedge (p \neq r \Rightarrow p.L = (p.P).L + 1)$
 $GoodL(p)$       $\equiv (\forall q \in Neig_p :: |p.L - q.L| > 1 \Rightarrow (p.L < q.L \vee q.S = E))$
 $GP\text{-}REP(p)$     $\equiv (\exists q \in Neig_p :: q.S = E \vee q.L - p.L > 1)$
 $Start(p)$       $\equiv p.Req = OUT \wedge GP\text{-}REP(p)$
 $End(p)$         $\equiv p.Req = REP \wedge \neg GP\text{-}REP(p)$
..........................................................................
Algorithm for $p = r$:
  Constants: $p.S = C; p.P = \perp; p.L = 0$;
  Predicates:
    $Allowed(p) \equiv true$
  Actions:
    $A\text{-}action :: Start(p) \rightarrow p.Req := ASK$;
    $O\text{-}action :: End(p)   \rightarrow p.Req := OUT$;
Algorithm for $p \neq r$:
  Variables: $p.S \in \{C, E\}; p.P \in Neig_p; p.L \in \mathbb{N}$;
  Predicates:
    $Allowed(p)$          $\equiv false$
    $AbnormalTree(p)$     $\equiv p.S = C \wedge ((p.P).S = E \vee (p.P).L \geq p.L)$
    $Connect(p)$          $\equiv (\exists q \in Neig_p :: q.Req = REP \wedge q = MinChPar(p)$
                          $\wedge (p.S = C \Rightarrow p.L - q.L > 1))$
  Actions:
    $E\text{-}action :: AbnormalTree(p)$ $\rightarrow p.S := E$;
    $C\text{-}action :: Connect(p)$      $\rightarrow p.S := C; p.P := MinChPar(p); p.L := (p.P).L + 1$;
                                   $p.Req := OUT$;
    $A\text{-}action :: Start(p)$        $\rightarrow p.Req := ASK$;
    $O\text{-}action :: End(p)$          $\rightarrow p.Req := OUT$;
```

When a BFS tree is constructed, the following property is verified at each processor $p \in V, p \neq r$: The level of p's parent is equal to p's level minus 1 (i.e., $(p \neq r) \Rightarrow (p.L = (p.P).L + 1)$). For processor r, we have the following constant values: r has no parent and a level equal to zero (i.e., $(p = r) \Rightarrow (p.P = \perp \wedge p.L = 0)$). Moreover, according to Claim 3 of Definition 1 we must have that the deviation on the level values between any processor $p \in V$ and its neighbors does not exceed one (i.e., $\forall q \in Neig_p, |q.L - p.L| < 1$). If one of these above constraints are not verified then a BFS tree is not constructed. Therefore, we have either at least one abnormal tree in $\mathcal{F}$ or there is a processor $p \in V$ with a neighbor q such that $q.L - p.L > 1$ (i.e., Predicate $GP\text{-}REP(p)$ is satisfied at p). In these cases, processor p executes $A\text{-}action$ to set the shared variable $p.Req$ to ASK in order to ask the permission to allow q to connect to p, if p is not already asking a permission (i.e., we have $p.Req = OUT$). To this end, Algorithm 2 sends a request to the root of the tree.

Inputs for Algorithm 2. In order to allows Algorithm 2 to send a request the following inputs are given at processor p: (i) $Child(p)$ is the set of children of p in the tree (i.e., $Child(p) \equiv \{q \in Neig_p : q.P = p\}$), (ii) $Parent(p)$ is the parent of p in the tree (i.e., $Parent(p) \equiv p.P$), (iii) $Height(p)$ is the height in the tree of the requesting processor p, and (iv) $Allowed(p)$ is a predicate which notifies if p can deliver permissions (i.e., $Allowed(p) \equiv (p = r)$). Remind that $Allowed(p)$ must be satisfied only at processor $p = r$ in Algorithm 2 to allow that eventually every processor joins the tree rooted at r, since eventually the processors cannot join another tree in forest $\mathcal{F}$.

In the case a permission is delivered at processor p (i.e., we have $p.Req = REP$), then each neighbor q of p can execute $C\text{-}action$ to connect to p. However to construct a BFS tree without an overcost on moves, processor q waits for until its neighbor x with the smallest level in a normal tree gives its authorization to q to connect by executing $C\text{-}action$ (i.e., we have $x.Req = REP \wedge x = MinChPar(q)$). When processor q executes $C\text{-}action$ then it sets its variables $p.P$ and $p.L$ according to its new parent in the tree, and it changes its status to Status C and its shared variable $p.Req$ to OUT. Finally, if there is no neighbor for which processor p needs a permission (i.e., Predicate $GP\text{-}REP(p)$ is no more satisfied at p), then p executes $O\text{-}action$ to set its shared variable $p.Req$ to OUT. This informs Algorithm 2 that the permission can be removed at p, then this allows p to ask a new permission later.

4 Question-Answer problem

In this section, we present a snap-stabilizing algorithm to implement the oracle used by the BFS tree construction given in Section 3. Formally, this oracle has to solve the Question-Answer problem which can be stated as following, a formal specification is given in Specification 2.

Given a static forest $\mathcal{F}$ of trees in a network $G = (V, E)$, a set of processors $De \subseteq V$ requesting a permission to make a defined computation and a set of processors $AP \subset V$ authorized to deliver permissions. Each $p \in AP$ is a root of a tree $T \in \mathcal{F}$. The *Question-Answer* problem is to deliver a permission (or *acknowledgement*) to a processor p in a tree $T \in \mathcal{F}$ if and only if the root q of T is in AP.

Specification 2 (Question-Answer). *Let $G = (V, E)$ be a network and $\mathcal{F}$ the static forest of trees in G. Let a tree $T \in \mathcal{F}$ and $root(T)$ the root of T. T is an* allowed tree *if $root(T) \in AP$ and* not allowed *otherwise. A protocol P which resolves the Question-Answer problem satisfies:*

[Liveness 1] *During an infinite computation, if a processor has to send infinitely often a request and it cannot send its request in an allowed tree, then there exist an infinite number of requests which were sent.*

[Liveness 2] *For every computation suffix, if a processor in an allowed tree has sent a request at time t, then there exist at least one processor in the same tree which receives an acknowledgement to its own sent request at time $t' > t$.*

[Safety 1] *Every processor which has sent a request receives at most one acknowledgement causally related to its sent request.*

[Safety 2] *Every processor in a not allowed tree which has sent a request never receives an acknowledgement.*

Remark that only semi-algorithms can satisfy Specification 2, that is no acknowledgement is sent to processors in a not allowed tree, from Property [Safety 2] of Specification 2.

4.1 Question-Answer Algorithm

In this section, we present a snap-stabilizing algorithm for the Question-Answer problem, a formal description is given by Algorithm 2. This is a non-uniform algorithm because some rules are only executed by a subset of processors $p \in V$ satisfying a local Predicate $Allowed(p)$ (i.e., p can deliver a permission or not).

Variables. We define below the different variables used by Algorithm 2.

Shared variable. Each processor $p \in V$ has a local shared variable $p.Req$ which allows an external algorithm to require the Question-Answer algorithm at p. This shared variable can take four values: ASK, $WAIT$, REP, and OUT. By setting the shared variable $p.Req$ to ASK in the external algorithm, p requests a permission through the Question-Answer algorithm to its root of the tree. To this end, Question-Answer algorithm tries to send a request to the root of the tree and sets the shared variable $p.Req$ to $WAIT$. At least the request of a requesting processor with the lowest level (or height) in the tree will reach the root and then receive a permission (an *acknowledgement*). When p receives an acknowledgement, it sets $p.Req$ to REP. Finally, the external algorithm must set $p.Req$ to OUT to request another permission through Question-Answer algorithm.

Local variables. Each processor $p \in V$ maintains two local variables:

- $p.Q$: it defines the status of the Question-Answer algorithm at processor p. There are three distinct status: R, W, and A. Status R notifies that p transmits a request to the root of the tree, whereas Status W indicates that p waits for an acknowledgement from the root for the transmitted request. The third status, Status A, indicates that p has received an acknowledgement from the root.
- $p.HQ$: it stores at p the height of the processor which has sent the request.

Algorithm Description. To simplify the presentation of the algorithm, consider a forest of allowed trees (i.e., trees rooted at nodes p satisfying Predicate $Allowed(p)$) and a fixed set of requests. In the following, we explain the way our algorithm handles requests focusing on a single tree T of the forest, but this is the same for other trees since

Algorithm 2. Question-Answer algorithm for any $p \in V$

```
Inputs: $Neig_p$: set of (locally) ordered neighbors of $p$;
        $Child(p)$: set of neighbors considered as children of $p$ in the tree;
        $Allowed(p)$: predicate which indicates if $p$ is able to acknowledge to a request;
        $Parent(p)$: parent of $p$ in the tree, equal to a processor $q \in Neig_p$ if $\neg Allowed(p)$ or equal to $\perp$ otherwise ;
        $Height(p)$: height of $p$ in the tree;
Shared variable: $p.Req \in \{ASK, WAIT, REP, OUT\}$;
Variables: $p.Q \in \{R, W, A\}$; $p.HQ \in \mathbb{N}$;
.............................................................................
Macros:
 $RC(p) = \{q \in Child(p) :: q.Q \in \{R, W\}\}$
 $PrioRC(p) = \{q \in RC(p) :: \forall t \in RC(p), q.HQ \leq t.HQ\}$
 $Ch_p = \min\{q \in PrioRC(p)\}$
.............................................................................
Global Predicates:
 $Transmit(p) \equiv p.Q = A \wedge (\forall q \in Child(p) :: q.Q = W \Rightarrow q.HQ \neq p.HQ)$
 $Retransmit(p) \equiv p.Q = W \wedge (\exists q \in Child(p) :: q.Q = R \wedge q.HQ = p.HQ)$
 $Error(p) \equiv p.Q \neq A \wedge [(p.Req \notin \{ASK, WAIT\} \wedge p.HQ = Height(p)) \vee (p.HQ \neq Height(p)$
     $\wedge (p.Req \neq REP \Rightarrow (\forall q \in Child(p) :: q.HQ = p.HQ \Rightarrow q.Q = A)))]$
 $Request(p) \equiv p.Req = ASK \wedge (|PrioRC(p)| > 0 \Rightarrow Height(p) \leq (Ch_p).HQ)$
 $RequestT(p) \equiv p.Req \neq REP$
     $\wedge |PrioRC(p)| > 0 \wedge [((Ch_p).HQ \geq p.HQ \Rightarrow Transmit(p)) \vee Retransmit(p)]$
.............................................................................
Algorithm:
  Predicates:
  $Wait(p) \equiv (Allowed(p) \wedge p.Q = R \wedge (\forall q \in Child(p) :: q.HQ = p.HQ \Rightarrow q.Q = W)) \vee$
      $(\neg Allowed(p) \wedge Parent(p).Q = R \wedge p.Q = R \wedge Parent(p).HQ = p.HQ$
      $\wedge (\forall q \in Child(p) :: q.HQ = p.HQ \Rightarrow q.Q = W))$
  $Answer(p) \equiv (Allowed(p) \wedge p.Q = W) \vee$
      $(\neg Allowed(p) \wedge Parent(p).Q = A \wedge p.Q = W \wedge Parent(p).HQ = p.HQ)$
  Actions:
  $QE$-action  :: $Error(p)$    $\rightarrow$ $p.Q := A; p.HQ := Height(p)$;
  $QR$-action  :: $Request(p)$  $\rightarrow$ $p.Q := R; p.HQ := Height(p); p.Req = WAIT$;
  $QRC$-action :: $RequestT(p)$ $\rightarrow$ $p.Q := R; p.HQ := (Ch_p).HQ$;
                                    if $p.HQ < Height(p) \wedge p.Req = WAIT$ then $p.Req := ASK$; fi
  $QW$-action  :: $Wait(p)$     $\rightarrow$ $p.Q := W$;
  $QA$-action  :: $Answer(p)$   $\rightarrow$ $p.Q := A$;
                                    if $p.Req = WAIT$ then $p.Req := REP$; fi
```

the requests in each tree are handled independently. In the algorithm, the requests sent by nodes of lowest height in the tree are handled in priority.

When a processor p has a *local request* requested by the external algorithm (i.e., $p.Req = ASK$), p can execute QR-*action* to set its variables $p.Req, p.Q$, and $p.HQ$ to $WAIT, R$, and to $Height(p)$ respectively, in order to send its request to the root of the tree it belongs to. The external algorithm is informed that the request is sent since $p.Req = WAIT$. Otherwise, an internal processor p in the tree with no local request (i.e., $p.Req \neq REP$) could have to transmit requests from its children (the request from a requesting descendant of lowest height first) in the following cases:

– a child of p is sending a request with a highest priority (i.e., $(Ch_p).HQ < p.HQ$);
– the acknowledgement received for the transmitted request is no more needed at p (all its children waiting it have transfered the acknowledgment, see Predicate $Transmit(p)$);
– p is waiting for an acknowledgement for a request and a new request is transmitted by a child of p with the same height (see Predicate $Retransmit(p)$).

In all these above cases, p executes QRC-*action* to set $p.Q$ to R and $p.HQ$ to the lowest height among requesting descendant of p (i.e., $p.HQ = (Ch_p).HQ$).

A processor p waits for an acknowledgement for a current request when its parent has transmitted the request (see Predicate $Wait(p)$). Moreover, all p's children transmitting the same request (i.e., with the same height) have to wait for an acknowledgement. Hence, Status W allows to remove bad requests due to an incorrect initial configuration and to synchronize request transmissions of same priority. In this case, p sets its variable $p.Q$ to W using $QW\text{-}action$.

When the root $root(T)$ of the tree T has no local request and is waiting for an acknowledgement for requesting descendant(s) (see Predicate $Answer$), then it executes $QA\text{-}action$ to set its variable $root(T).Q$ to A. This permission is propagated down in the tree to the requesting descendant(s) following the path(s) used to transmit the request. Finally, a processor p waiting for an acknowledgement to a local request (i.e., $p.Q = W$ and $p.Req = WAIT$) executes $QA\text{-}action$ to receive the acknowledgement and sets the shared variable $p.Req$ to REP to notify to the external algorithm of the delivered permission. Note that as soon as a received acknowledgement is no more needed at a processor p (i.e., $p.Req$ is setted to OUT by the external algorithm), then another request transmitted by a child of p can be transmitted by p up in the tree.

However, a processor must be able to detect *wrong* requests due to an incorrect initial configuration. A request treated by a processor p is a *wrong request* in the following cases (see Predicate $Error(p)$):

- p is sending a local request whereas it has no local request (i.e., $p.Q \neq A \wedge p.Req \notin \{ASK, WAIT\}$ and $p.HQ = Height(p)$);
- p is transmitting a request from a child, however no child of p has a request with the same height (i.e., $p.Q \neq A \wedge p.HQ \neq Height(p) \wedge (\forall q \in Child(p), q.HQ = p.HQ \Rightarrow q.Q = A)$).

When a processor p detects a wrong request, then p executes $QE\text{-}action$. This action has the highest priority among the actions at p, and it reinitiates p's state like if an acknowledgement to a local request was received, i.e., to set $p.Q$ to A and $p.HQ$ to $Height(p)$ (without changing the state of the shared variable $p.Req$).

A questioning mechanism close to the mechanism presented here was used in [12] to design a snap-stabilizing solution to the problem of Propagation of Information with Feedback (PIF) with a round complexity in $O(n)$ and a step complexity in $O(\Delta n^3)$. However, solving the PIF problem involves a strong synchronization in the network to insure that all the nodes in the network belong to the same broadcast tree before to initiate the feedback phase. Indeed, each time a node is added to the broadcast tree the questioning mechanism is reset leading to a $O(n)$ round complexity. Contrary to this questioning mechanism, here our mechanism needs a weakest synchronization to resolve the Question-Answer problem. Let De the set of requesting nodes and h_{min} the height of closest requesting nodes from the root in T. The first requests acknowledged by $root(T)$ are the requests from nodes at height h_{min}. Then, if the set of requests is static then the requests at height $h_{min} + 1$ are acknowledged by $root(T)$ (if any) and so on. In fact, only a synchronization for the requests of requesting nodes at height h_{min} (whose requests are of highest priority) in tree T is required leading to a round complexity function of the height of T. The transmission of a request requires $O(n)$ steps, however this transmission can be interrupted only by a requesting node with the same height in T, that is at most $|De|$ times.

The following lemma summarizes the above discussion:

Lemma 1. *Let T an allowed tree and h_{min} the height of closest requesting nodes in De from the root in T, in $O(h_{min})$ rounds and $O(n|De|)$ steps at least one requesting node in De receive an acknowledgement from $root(T)$ to its request.*

5 Composition and Complexities

Algorithm $\mathcal{BFS}$ is obtained by composition of Algorithm 2 and Algorithm 1. These two algorithms are composed together at each processor $p \in V$ with a conditional composition (first introduced in [14]): Algorithm 1 $\circ\ |_{Cond(p)}$ Algorithm 2, where each guard g of the actions of Algorithm 2 at each processor $p \in V$ has the form $Cond(p) \wedge g$ with Predicate $Cond(p)$ defined below (see Algorithm 1 for the description of predicates): $Cond(p) \equiv GoodT(p) \wedge GoodL(p)$.

Using this composition, each processor $p \in V$ can execute Algorithm 2: (i) to transmit requests and acknowledgements only if the tree containing p is locally correct (i.e., Predicate $GoodT(p)$ is satisfied), and (ii) to ask a permission if needed (i.e., Predicate $GoodL(p)$ is satisfied). Indeed, actions in Algorithm 2 can be locked to avoid processors belonging to a tree not rooted at r (abnormal tree) to transmit useless requests since no acknowledgement can be received (only r can deliver acknowledgements). Therefore, processors in abnormal trees can only execute actions in Algorithm 1 to hook on to another tree in the forest via a neighbor with a permission (acknowledgement delivered by Algorithm 2). Moreover, actions of Algorithm 2 and Algorithm 1 can be enabled at p simultaneously. In this case, Algorithm 2 is executed before Algorithm 1 at p.

Algorithm $\mathcal{BFS}$ uses Algorithm 2 which can be viewed as a synchronizer allowing the BFS tree construction of T rooted at r layer by layer, the addition of any new layer of processors depending of a permission request. The requesting processors closest to r at height k in T receive an acknowledgement to their request from r in $O(k)$ rounds (Lemma 1) which allows their neighbors to hook on to T. The same argument holds for the addition of each new layer of T. Moreover, the height of a BFS tree is lower than or equal to the network diameter. Therefore, summing up the round complexity associated to each layer we obtain a round complexity $O(d^2)$ to construct a BFS tree, with d the network diameter. In another hand, the mechanism we use for deleting the abnormal trees is obviously in $O(n)$ rounds, since the height of such a tree can be in $O(n)$. But any processor in an abnormal tree far from the root of this tree will become the neighbor of at least a processor of the normal BFS tree in $O(d^2)$ rounds and will hook to it even if the abnormal tree is not yet deleted. So the global round complexity is still $O(d^2)$ as stated in the following lemma.

Lemma 2. *From any configuration, in $O(d^2)$ rounds Algorithm $\mathcal{BFS}$ reaches a configuration $\gamma \in \mathcal{C}$ satisfying Definition 1, with d the diameter of the network.*

We discuss above the ideas leading to the round complexity of Algorithm $\mathcal{BFS}$. We give below the main arguments allowing to show that Algorithm $\mathcal{BFS}$ has a step complexity in $O(mn^4)$. We define a *topological change* as follows: Given a forest $\mathcal{F}$ of trees in a configuration $\gamma \in \mathcal{C}$, a *topological change* in $\mathcal{F}$ is obtained by the execution of E-*action* or C-*action* at a processor $p \in V$ in step $\gamma \mapsto \gamma'$. We first consider the step complexity of Algorithm 1. A processor can hook on to several abnormal trees until belonging to the tree rooted at r. First of all, we establish the number of connections to an abnormal tree that any processor $p \in V$ can make until belonging to the tree rooted at r. In the reminder, the tree rooted at r is noted $Tree(r)$ and $root(T)$ describes the root node of a tree T.

Proposition 1. *Every processor $p \in V$ is hooked on to the neighbor q such that $\forall s \in Neig_p, q.L \leq s.L$.*

Proof. According to the formal description of Algorithm 1, a processor hooks on to a neighbor using $C\text{-}action$. Assume, by the contradiction, that there is a processor $p \in V$ such that $\exists s \in Neig_p, (p.P).L > s.L$. We must consider two cases: s is in an abnormal tree or not. If s is in an abnormal tree then either $s.S = E$ then $s \notin MinChPar(p) \Rightarrow \neg Connect(p)$ a contradiction, or $s.S = C$ then by Property [Safety 2] of Specification 2 s never receives an acknowledgement and we have that $s.Req \neq REP \Rightarrow \neg Connect(p)$, otherwise $C\text{-}action$ is enabled at p, a contradiction. If s is in a normal tree then by Property [Liveness 2] of Specification 2 we have that $s.Req = REP$ and $C\text{-}action$ is enabled at p, a contradiction. □

Lemma 3. *Let any abnormal tree $T \in \mathcal{F}$ and the set of processors $B = \{p \in V : p \notin T \wedge (\exists q \in Neig_p :: q \in T)\}$. In any execution, only processors in B can hook on to T.*

Proof. Consider any abnormal tree $T \in \mathcal{F}$ in configuration $\gamma \in \mathcal{C}$. According to the formal description of Algorithm 1, a processor p must execute $C\text{-}action$ to hook on to a tree, i.e., there is a neighbor q such that $q.Req = REP$. Suppose that every processor $q \in B$ executes $C\text{-}action$ and they are hooked on to T in configuration γ_k. Note that after executing $C\text{-}action$, we have $q.Req = OUT$ at every processor $q \in B$. Assume, by the contradiction, that there is a processor $p \notin T$ in configuration γ_k which hooks on to T in step $\gamma_k \mapsto \gamma_{k+j}, j > 0$. This implies that p hooks on to a neighbor $q \in B$ (by definition of B) such that $q.Req = REP$, a contradiction by Property [Safety 2] of Specification 2 because q cannot receive an acknowledgement from $root(T)$ since T is an abnormal tree. □

Proposition 2. *Let a processor $p \in V$ which hooks on to a tree T in configuration $\gamma_i \in \mathcal{C}$. If another processor $q \in V$ hooks on to T by p in $\gamma_{i+j}, j > 0$, then T is a normal tree.*

Proof. According to Lemma 3, the expansion of an abnormal tree T' is limited at distance one from T'. After p hooks on to T, to allow the processor q to hook on to T by p then p receives an acknowledgement from $root(T)$. Therefore, T is a normal tree by Specification 2. □

Lemma 4. *Let any abnormal tree $T \in \mathcal{F}$. A processor $p \in V$ can hook on to T at most once by the same neighbor $q \in T$.*

Proof. Assume, by the contradiction, that there is a configuration $\gamma_k \in \mathcal{C}$ such that there is a processor $p \in V$ which hooks on to T by the same neighbor $q \in T$ a second time. To hook on to T, p must execute $C\text{-}action$, i.e., there is a neighbor $x \in T$ of p such that $x.S = C$ and $x.Req = REP$. According to Proposition 1, p hooks on to the neighbor $x \in V$ such that $x.S = C \wedge (\forall s \in Neig_p, x.L \leq s.L)$. Suppose that

p hooks on to T by the neighbor q a first time in step $\gamma_{i-1} \mapsto \gamma_i \in \mathcal{C}$, then p hooks on to another neighbor s of p, $s \neq q$, in step $\gamma_{j-1} \mapsto \gamma_j \in \mathcal{C}, j > i$. Now, we must consider several cases in configuration $\gamma_k, i < j < k$. If p is hooked on to s in γ_j because $q.S = E$ and $s.Req = REP$ in γ_i then since $q \in T$ we have $q.S = E$ in γ_k and $q \notin MinChPar(p) \Rightarrow \neg Connect(p)$, a contradiction. Otherwise $s.S = q.S = C$ and p is hooked on to s in $\gamma_j, i < j < k$, because $s.L < q.L$ and $s.Req = REP$. When p hooks on to q the first time in step $\gamma_{i-1} \mapsto \gamma_i$, we have $s.S = E$ or $s.L > q.L$. Since we have $s.S = C \wedge s.L < q.L \wedge s.Req = REP$ and p hooks on to s in step $\gamma_{j-1} \mapsto \gamma_j$, this implies that s is in a normal tree in γ_j according to Proposition 2. Thus, we have $s.S = C \wedge s.L < q.L$ in γ_k and $q \notin MinChPar(p) \Rightarrow \neg Connect(p)$, a contradiction. □

Lemma 5. *In any execution, every processor $p \in V \backslash \{r\}$ produces at most 2Δ topological changes in forest $\mathcal{F}$ while $p \notin Tree(r)$, with Δ the maximum degree of a processor in the network.*

Proof. To hook on to a tree, a processor $p \in V$ must execute $C\text{-}action$. According to Lemma 4, p cannot hook on to an abnormal tree $T \in \mathcal{F}$ twice by the same neighbor q of p. Since a processor can have at most Δ neighbors, p can hook on at most Δ times to an abnormal tree. Observe that $E\text{-}action$ has a higher priority than $C\text{-}action$ and $E\text{-}action$ can be executed between two executions of $C\text{-}action$, i.e., at most Δ times while $p \notin Tree(r)$. Therefore, by the definition of a topological change the lemma follows. □

After giving a bound for the number of connections to abnormal trees, we provide below an upper bound for the number of connections that a processor can make in the normal tree.

Remark 1. *For every processor $p \in Tree(r)$, $E\text{-}action$ is disabled at p.*

Lemma 6. *In any execution, every processor $p \in V \backslash \{r\}$ produces at most n topological changes in forest $\mathcal{F}$ while $p \in Tree(r)$, with n the number of processors in the network.*

Proof. Observe that for every processor $p \in Tree(r)$ we have $p.S = C$. Moreover, by Remark 1, for every processor $p \in Tree(r)$, $E\text{-}action$ is disabled. So, by definition the only topological change in $\mathcal{F}$ that a processor $p \in Tree(r)$ can produce is to execute $C\text{-}action$ in order to reduce its level in $Tree(r)$. Thus, by Proposition 1 each execution of $C\text{-}action$ by a processor $p \in Tree(r)$ in step $\gamma_i \mapsto \gamma_{i+1}$ implies that p hooks on to the neighbor with the lowest level in γ_{i+1} and $p.L$ in γ_i is higher than $p.L$ in γ_{i+1}. Therefore, since the size of $Tree(r)$ is bounded by n then any processor p can hook on to at most $n - 1$ processors by executing $C\text{-}action$ while $p \in Tree(r)$. □

From the above lemmas, each processor can hook on to at most $2\Delta + n$ times until reaching its correct position in the final BFS tree. Thus, there are at most $2\Delta n + n^2$ topological changes in the forest until a BFS tree is reached. Moreover, each topological change yields at most Δ requests in the network, so to construct a BFS tree at most

$2\Delta m + mn$ requests are generated by Algorithm 1. We now consider the step complexity of Algorithm 2. According to Lemma 1, a processor receives an acknowledgement with Algorithm 2 in $O(n^2)$ steps (since $|De| \leq n$). So, in $O(n^3)$ steps every requesting processor receives an acknowledgement (there are at most n requests in the network).

Given an upper bound on the number of requests generated by Algorithm 1, we have to multiply this amount by the number of steps needed by Algorithm 2 to acknowledge these requests in order to obtain an upper bound to the total step complexity of Algorithm $\mathcal{BFS}$. This is stated by the following lemma.

Lemma 7. *From any configuration, $O(\Delta mn^3 + mn^4)$ steps are needed by Algorithm $\mathcal{BFS}$ to reach a terminal configuration.*

Notice that in one hand using a questioning mechanism allows us to save steps by avoiding the transmission of useless requests, but in the other hand we obtain a higher round complexity ($O(d^2)$ instead of $O(d)$ with standard algorithms for BFS trees) due to the fact that permissions must be delivered before the add of new nodes to the constructed tree. Moreover, the step complexity established in Lemma 7 is not related with any initial value of a variable and it holds under any fairness assumptions.

Lemma 7 implies that Algorithm $\mathcal{BFS}$ always satisfies Property [TC1] of Specification 1 and is silent. We now consider any terminal configuration. Since the configuration is terminal, no action is enabled in Algorithm $\mathcal{BFS}$. It is trivial to verify by induction on the distance of a processor to r that every processor is in $Tree(r)$ and at the right level as stated in the following lemma.

Lemma 8. *Every terminal configuration reached by Algorithm $\mathcal{BFS}$ satisfies Definition 1.*

According to Lemmas 7 and 8, Algorithm $\mathcal{BFS}$ always satisfies respectively Properties [TC1] and [TC2] of Specification 1. Therefore, we can state the following theorem.

Theorem 1. *Algorithm $\mathcal{BFS}$ is a silent snap-stabilizing algorithm to construct a BFS tree.*

6 Conclusion

In this paper a silent snap-stabilizing algorithm resolving the Question-Answer problem has been given, in which each node requests a permission (delivered by a subset of network nodes) in order to perform a defined computation. Based on this algorithm, the first fully polynomial stabilizing algorithm for the construction of a spanning tree has been presented. A Breadth First Search tree is constructed in $O(d^2)$ rounds and in $O(n^6)$ steps, with d the diameter and n the number of nodes in the network. Moreover, a distributed daemon without any fairness assumptions is considered.

One crucial open question is the following: Is it possible to design a fully polynomial self-stabilizing algorithm to construct a spanning tree in $O(d)$ rounds with a polynomial step complexity?

References

1. Afek, Y., Kutten, S., Yung, M.: Memory-Efficient Self-Stabilizing Protocols for General Networks. In: van Leeuwen, J., Santoro, N. (eds.) WDAG 1990. LNCS, vol. 486, pp. 15–28. Springer, Heidelberg (1991)

2. Arora, A., Gouda, M.: Distributed reset (extended abstract). In: Veni Madhavan, C.E., Nori, K.V. (eds.) FSTTCS 1990. LNCS, vol. 472, Springer, Heidelberg (1990)
3. Awerbuch, B., Kutten, S., Mansour, Y., Patt-Shamir, B., Varghese, G.: Time optimal self-stabilizing synchronization. In: 25th Annual ACM Symposium on Theory of Computing (STOC), pp. 652–661 (1993)
4. Bui, A., Datta, A.K., Petit, F., Villain, V.: State-optimal snap-stabilizing pif in tree networks. In: Workshop on Self-stabilizing Systems (WSS), pp. 78–85. IEEE Computer Society (1999)
5. Burman, J., Kutten, S.: Time Optimal Asynchronous Self-Stabilizing Spanning Tree. In: Pelc, A. (ed.) DISC 2007. LNCS, vol. 4731, pp. 92–107. Springer, Heidelberg (2007)
6. Chen, N.-S., Yu, H.-P., Huang, S.-T.: A self-stabilizing algorithm for constructing spanning trees. Inf. Process. Lett. 39(3), 147–151 (1991)
7. Collin, Z., Dolev, S.: Self-stabilizing depth-first search. Inf. Process. Lett. 49(6), 297–301 (1994)
8. Cournier, A.: Mémoire d'Habilitation à Diriger les Recherches: Graphes et algorithmique distribuée stabilisante. Université de Picardie Jules Verne (2009)
9. Cournier, A.: A New Polynomial Silent Stabilizing Spanning-Tree Construction Algorithm. In: Kutten, S., Žerovnik, J. (eds.) SIROCCO 2009. LNCS, vol. 5869, pp. 141–153. Springer, Heidelberg (2010)
10. Cournier, A., Devismes, S., Petit, F., Villain, V.: Snap-stabilizing depth-first search on arbitrary networks. The Computer Journal 49(3), 268–280 (2006)
11. Cournier, A., Devismes, S., Villain, V.: A Snap-Stabilizing DFS with a Lower Space Requirement. In: Tixeuil, S., Herman, T. (eds.) SSS 2005. LNCS, vol. 3764, pp. 33–47. Springer, Heidelberg (2005)
12. Cournier, A., Devismes, S., Villain, V.: Snap-stabilizing pif and useless computations. In: 12th International Conference on Parallel and Distributed Systems (ICPADS), pp. 39–48. IEEE Computer Society (2006)
13. Cournier, A., Devismes, S., Villain, V.: Light enabling snap-stabilization of fundamental protocols. ACM Transactions on Autonomous and Adaptive Systems (TAAS) 4(1) (2009)
14. Datta, A.K., Gurumurthy, S., Petit, F., Villain, V.: Self-stabilizing network orientation algorithms in arbitrary rooted networks. Stud. Inform. Univ. 1(1), 1–22 (2001)
15. Datta, A.K., Larmore, L.L., Vemula, P.: Self-Stabilizing Leader Election in Optimal Space. In: Kulkarni, S., Schiper, A. (eds.) SSS 2008. LNCS, vol. 5340, pp. 109–123. Springer, Heidelberg (2008)
16. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Commun. ACM 17(11), 643–644 (1974)
17. Dolev, S.: Self-Stabilization. MIT Press (2000)
18. Dolev, S., Israeli, A., Moran, S.: Self-stabilization of dynamic systems assuming only read/write atomicity. In: 9th ACM Symposium on Principles of Distributed Computing (PODC), pp. 103–117 (1990)
19. Gärtner, F.: A survey of self-stabilizing spanning-tree construction algorithms. Tech. rep., EPFL (October 2003)
20. Huang, S.-T., Chen, N.-S.: A self-stabilizing algorithm for constructing breadth-first trees. Inf. Process. Lett. 41(2), 109–117 (1992)
21. Johnen, C.: Memory-efficient self-stabilizing algorithm to construct bfs spanning trees. In: 3rd Workshop on Self-stabilizing Systems (WSS), pp. 125–140 (1997)
22. Kosowski, A., Kuszner, Ł.: A Self-Stabilizing Algorithm for Finding a Spanning Tree in a Polynomial Number of Moves. In: Wyrzykowski, R., Dongarra, J., Meyer, N., Waśniewski, J. (eds.) PPAM 2005. LNCS, vol. 3911, pp. 75–82. Springer, Heidelberg (2006)
23. Tel, G.: Introduction to distributed algorithm, 2nd edn. Cambridge University Press (2000)