Fast and Compact Distributed Verification and Self-Stabilization of a DFS Tree

Shay Kutten*    Chhaya Trehan†

August 5, 2014

Abstract

We present algorithms for distributed verification and silent-stabilization of a DFS (Depth First Search) spanning tree of a connected network. Computing and maintaining such a DFS tree is an important task, e.g., for constructing efficient routing schemes. Our algorithm improves upon previous work in various ways. Comparable previous work has space and time complexities of $O(n \log \Delta)$ bits per node and $O(nD)$ respectively, where $\Delta$ is the highest degree of a node, $n$ is the number of nodes and $D$ is the diameter of the network. In contrast, our algorithm has a space complexity of $O(\log n)$ bits per node, which is optimal for silent-stabilizing spanning trees and runs in $O(n)$ time. In addition, our solution is modular since it utilizes the distributed verification algorithm as an independent subtask of the overall solution. It is possible to use the verification algorithm as a stand alone task or as a subtask in another algorithm. To demonstrate the simplicity of constructing efficient DFS algorithms using the modular approach, We also present a (non-silent) self-stabilizing DFS token circulation algorithm for general networks based on our silent-stabilizing DFS tree. The complexities of this token circulation algorithm are comparable to the known ones.

1 Introduction

A clear separation is common between the notions of computing and verification in sequential systems. A similar separation in the context of distributed systems has been emerging. Distributed verification of global properties like minimum spanning trees have been devised [20].

An area of distributed systems that can greatly benefit from this separation is that of self-stabilization. Self-stabilization is the ability of a system to recover from transient faults. A self-stabilizing distributed system can be started in

*email: kutten@ie.technion.ac.il. Faculty of Industrial Engineering and Management, Technion, Haifa, Israel.
†email: chhaya.dhingra@gmail.com. Faculty of Industrial Engineering and Management, Technion, Haifa, Israel.
any arbitrary configuration and must eventually converge to a desired legal behavior. Self-stabilizing algorithms can run a distributed verification algorithm repeatedly to detect the occurrence of faults in the system and take the necessary action for convergence to a legal behavior. This is the approach we take here in devising a silent-stabilizing DFS algorithm. The concept of first detecting a fault and then taking the corrective measures for self-stabilization was first introduced by [19], [1] and [3]. The approach taken by Katz and Perry in [19] is that of global detection of faults by a leader node that periodically takes the snapshots of the global state of the network and resets the system if a fault is detected. Afek, Kutten and Yung [1], and Awebuch et al [3] on the other hand, suggested that the faults in the global state of a system could sometimes be detected by local means - i.e., by having each node check the states of all its neighbors. Korman, Kutten and Peleg further formalized the idea of local detection of faults in [21] and introduced the concept of proof labeling schemes. A proof labeling scheme works by assigning a label to every node in the input network. The collection of labels assigned to the nodes acts as a locally checkable distributed proof that the global state of the network satisfies a specific global predicate. A proof labeling scheme consists of a pair of algorithms \((M, V)\), where \(M\) is a marker algorithm that generates a label for every node and \(V\) is a verifier algorithm that checks the labels of neighboring nodes. In this paper, we present a proof labeling scheme for detecting faults in the distributed representation of a DFS spanning tree. For self-stabilization, the DFS tree is computed afresh and new labels are assigned to the nodes by the marker on detection of faults.

1.1 Additional Related Work

Dijkstra introduced the concept of Self-stabilization [11] in distributed systems. Self-stabilization deals with the faults that entail an arbitrary corruption of the state of a system. These faults are rather severe in nature but do not occur very frequently in reality [28]. Table 1.1 summarizes the known complexity results for self-stabilizing DFS algorithms. Collin and Dolev presented a silent-stabilizing DFS tree algorithm in [7]. Their algorithm works by having each node store its path to the root node in the DFS tree. Since the path of a node to the root in a DFS tree can be as long as \(n\), the number of nodes in the network, the space complexity of their algorithm is \(O(n \log \Delta)\) per node, where \(\Delta\) is the highest degree of a node in the network. The time complexity of their algorithm under the contention time model is \((n\Delta)\). We drop the multiplicative factor of \(\Delta\) from their time complexity here for the sake of comparison with all the other algorithms that do not count their time under the contention model. Cournier et al presented a snap-stabilizing DFS wave protocol in [8] which snap stabilizes with a space complexity of \(O(n + \log n)\).

Considerable work has been invested in developing self-stabilizing depth-first token circulation algorithms with multiple successive papers improving each other. All of these algorithms also generate a DFS tree in every token circulation
Table 1: Comparing self-stabilizing DFS algorithms

| Algorithm | Space     | Time       | Remarks                                      |
|-----------|-----------|------------|----------------------------------------------|
| [7]       | $O(n \log \Delta)$ | $O(nD)$   | Silent                                       |
| [8]       | $O(n \log n)$  | 0         | Snap Stabilizing                             |
|           |            |            | First DFS wave, needs Unique IDs            |
| [9]       | $O(\log n)$ | 0         | Snap Stabilizing                             |
|           |            |            | Wave takes $O(n^2)$ rounds                   |
| [10]      | $O(\log n)$ | $O(nD)$   | Token Circulation, not silent                |
| [16]      | $O(\log n)$ | $O(nD)$   | Token Circulation, not silent                |
| [23]      | $O(\log \Delta)$ | $O(nD)$ | Token Circulation, not silent                |
|           |            |            | Requires neighbor of neighbor info           |
| [17]      | $O(\Delta)$ | $O(nD)$   | Token Circulation, not silent                |
| [24]      | $O(\log \Delta)$ | $O(nD)$ | Token Circulation, not silent                |
| OUR RESULTS | $O(\log n)$ | $O(n)$    | Two algorithms: Silent and Token Circulation |
|           |            |            | Both with the same complexity               |

round, however these algorithms are not silent. Self-stabilizing depth-first token circulation on arbitrary rooted networks was first considered by Huang and Chen in [16]. Their algorithm stabilizes in $O(nD)$ time with a space complexity of $O(\log n)$ bits per node. Subsequently several self-stabilizing DFS token circulation algorithms [10, 18, 17, 24] were devised. All these papers worked on improving the space complexity of [16] from $O(\log n)$ to a function of $\Delta$, the highest degree of a node in the network. The time complexity of all of the above token circulation algorithms [16, 18, 17, 24] is $O(nD)$ rounds, which is much more than the time it takes for one token circulation cycle on a given network. Petit improved the stabilization time complexity of depth-first token circulation in [23] with a space complexity of $O(\log n)$ bits per node. Petit and Villain [26] presented the first self-stabilizing depth-first token circulation algorithm that works in asynchronous message passing systems.

1.2 Our Contribution

The main contribution of the current paper is a silent self-stabilizing DFS spanning tree algorithm. The space complexity of our algorithm is $O(\log n)$ bits per node. The only other silent-stabilizing DFS tree algorithm [7] has a space complexity of $O(n \log \Delta)$. Dolev et al [13] established a lower bound of $O(\log n)$ bits per node on the memory requirement of silent-stabilizing spanning tree algorithms. Thus, ours is the first memory optimal silent-stabilizing DFS spanning
tree algorithm. The silent-stabilizing DFS construction algorithm is designed in a modular way consisting of separate modules for fault detection and correction. The distributed verification module of this algorithm can be considered a contribution in itself.

Finally, we present a self-stabilizing depth-first token circulation algorithm on general networks which uses our silent-stabilizing DFS tree as a module of the overall algorithm. The space and time complexities of our token circulation algorithm are as good as the previously published work on fast self-stabilizing depth-first token circulation [23]. The algorithm presented is modular and is composed of simple components that come together to achieve known results for the problem in a simpler way.

1.3 Outline of the paper

In the next section (Section 2), we describe the model of distributed systems considered in this paper. This section also includes some basic definitions and notations. Section 3 addresses the distributed verification algorithm which acts as the Verifier $V$ of the proof labeling scheme. The Marker $M$ of the proof labeling scheme is presented in Section 4. Section 5 describes the technique used to make the algorithm self-stabilizing. Section 6 presents the correctness proofs and performance analysis. Section 7 describes a token circulation scheme based on the new silent-stabilizing DFS spanning tree.

2 Preliminaries

A distributed system is represented by a connected undirected graph $G(V, E)$ without self-loops and multiple edges, where each node $v \in V$ represents a processor in the network and each edge $e \in E$ corresponds to a communication link between its incident nodes. Processors communicate by writing into their own shared registers and reading from the shared registers of the neighboring processors. The network is assumed to be asynchronous. We do not require processors to have unique identifiers. We do assume the existence of a distinguished processor, called the root of the network. Each node $v \in V$ orders its edges by some arbitrary ordering $\alpha_v$ as in [7]. For an edge $(u, v)$, let $\alpha_u(v)$ denote the index of the edge $(u, v)$ in $\alpha_u$.

As opposed to Collin and Dolev [7], we use the (rather common) ideal time complexity which assumes that a node reads all of its neighbors in at most one time unit. Our results translate easily to an alternative, stricter, contention time complexity used by Collin and Dolev in [7], where a node can access only one neighbor in one time unit. The time cost of such a translation is at most a multiplicative factor of $\Delta$, the maximum degree of a node (it is not assumed that $\Delta$ is known to nodes). As is commonly assumed in the case of self-stabilization, each node has only some bounded number of memory bits available to be used. Here, this amount of memory is $O(\log n)$. 


Self-stabilization and silent-stabilization: A distributed algorithm is self-stabilizing if it can be started in any arbitrary global state and once started, the algorithm converges to a legal state by itself and stays in the legal state unless additional faults occur. A self-stabilizing algorithm is silent if starting from an arbitrary state it converges to a legal global state after which the values stored in the communication registers do not change, see e.g. [13]. While some problems like token circulation are non-silent by nature, many input/output algorithms allow a silent solution.

Spanning Tree: Distributed Representation: A spanning tree $T$ of a connected, undirected graph $G(V, E)$ is a tree composed of all the nodes and some of the edges of $G$. A spanning tree $T$ of some graph $G$ is represented in a distributed manner by having each node locally mark some of its incident edges such that the collection of marked edges of all the nodes forms a spanning tree of $G$. Actually, it is enough that each node marks its edge leading to its parent on the tree in a local variable.

DFS Tree and the first DFS Tree of a Graph A DFS Tree of a connected, undirected graph $G(V, E)$ is the spanning tree generated by a depth-first search (DFS) traversal of $G$. In a DFS traversal, starting from a specified node called the root, all the nodes of the graph are visited one at a time, exploring as far as possible before backtracking, see e.g. [15]. A first DFS traversal is the one that acts as follows: Every time there is a set of unexplored neighbors to choose from, the chosen edge is the edge with the smallest port number in the port ordering $\alpha_v$. The tree thus generated is called the first DFS tree [7]. While a connected, undirected graph can have more than one DFS spanning trees, it can have only one first DFS spanning tree.

Lexicographic Ordering A simple path from the root of a graph $G$ to some node $v \in V$ can be represented as a string starting with a $\perp$ followed by a sequence of the port numbers of the outgoing edges on the path [7]. Given such a string representation of a path between any two nodes, a lexicographic operator $\prec$ can be defined to compare multiple paths of a given node $v$ from the root, where $\perp$ is considered the minimum character. In the first DFS tree of a graph, the path of every node $v \in V$ from the root is the lexicographically smallest among all the simple paths from root to $v$ [7].

DFS Intervals In a DFS traversal, it is common to assign to each node an interval $(in, out)$ corresponding to the discovery and finish time of exploration of that node. The discovery time or $in$ is the time at which a node is discovered for the first time. The discovery time of a node $v \in V$ is denoted as $in_v$. The finish time of node $v$ denoted by $out_v$ is the time at which a node has finished exploring all its neighbors. These intervals have the property that given any two intervals $(in, out)$ and $(in', out')$, either one includes the other or they are totally disjoint. Assuming without loss of generality that $in < in'$, we can write this formally as: either $(in < in' < out' < out)$ or $(in < out < in' < out')$ [13]. In other words, the DFS intervals induce a partial order on the nodes of a graph.
2.1 Notation
We define the following notation to be used throughout:

- \( \eta(v) \) denotes the set of neighbors of \( v \) in \( G \). \( \forall v \in V \) \( (\eta(v) = \{ u | u \in V \land u \neq v \land (u, v) \in E \}) \).  
- \( interval_v \) denotes the \((in, out)\) label of \( v \).  
- \( in_v \) denotes the \( in \) label of \( v \) and \( out_v \) denotes the \( out \) label of \( v \).  
- Relational operator \( \subset \) between two intervals \((in, out)\) and \((in', out')\) indicates the inclusion of the first interval in the second one. For example: \((in, out) \subset (in', out')\) indicates that \((in, out)\) is included in \((in', out')\).  
- Relational operator \( \supset \) is defined similarly.

3 DFS Verification: **Verifier** \( V \)

Given a graph \( G(V, E) \) and the distributed representation of a spanning Tree \( T \) of \( G \), the DFS verification algorithm is required to verify that \( T \) is a DFS spanning tree of \( G \). The **Verifier** \( V \) takes as input a connected graph \( G(V, E) \) where each node \( v \in V \) bears an \((in, out)\) label in addition to \( v \)'s parent on \( T \). Note that \( V \) takes \((in, out)\) labels of nodes as input and is not concerned with where they came from. 

We assume that each node can look at the labels of all its neighbors in addition to its own label and state. A node cannot look at the state of any of its neighbors, however. Each node \( v \in V \) periodically checks the labels of all its neighbors and locally computes the following additional information from its own state and label as well as the labels of its neighbors.

3.1 Intermediate Computations

Each node computes the following **macros** to be used for verification.

1. There are zero or more neighbors of \( v \) whose interval includes \( v \)'s interval. Let us call the set of all such nodes the **neighboring ancestors** of \( v \) and denote this set by \( anc_l(v) \).

\[
anc_l(v) = \{ w | w \in \eta(v) \land interval_w \supset interval_v \}
\]

2. The parent of \( v \) as perceived by the labels: \( parent_l(v) = w | w \in anc_l(v) \land \forall u \in anc_l(v) (u \neq w \rightarrow interval_w \subset interval_u) \).

3. There are zero or more neighbors of \( v \) whose interval is included in \( v \)'s interval, let us call the set of all such nodes the **neighboring descendants** of \( v \) and denote this set by \( desc_l(v) \).

\[
desc_l(v) = \{ w | w \in \eta(v) \land interval_w \subset interval_v \}
\]
4. A child neighbor of $v$ is a neighboring descendant of $v$ whose interval is not included in the interval of any other neighboring descendant of $v$.

$$\text{child}_l(v) = \{u | u \in \text{desc}_l(v) \land \neg \exists u' \in \text{desc}_l(v) (u' \neq u \land \text{interval}_{u'} \supset \text{interval}_u)\}$$

5. $\text{children}_l(v) \subseteq \text{desc}_l(v)$ is the set of all child neighbors of $v$.

The subscript $l$ in $\text{anc}_l(v)$ above denotes that the set $\text{anc}_l(v)$ is computed by the node $v$ by just looking at the labels of $v$ and those of $v$’s neighbors. The same holds for all the other macros defined above. It is worth pointing out that all these are intermediate computations and the data they generate need not be stored on the node.

The verification is performed by having each node compute a set of predicates. If $T$ is indeed a DFS tree and the labels on all the nodes are proper (i.e. they are as if they were generated by an actual DFS Traversal of the input graph); then the Verifier accepts continuously on every node until a fault occurs. If a fault occurs either due to the corruption of the state of some nodes or due to some nodes having incorrect labels, at least one node rejects. The node that rejects is called a detecting node. The Verifier self-stabilizes trivially since it runs periodically.

### 3.2 Local Interval Predicates

Recall that each node encodes its parent on the input spanning tree $T$ by marking its edge leading to its parent in $T$ in a local variable. Let $\text{parent}_v$ denote the local variable used to store the parent of $v$ in $T$. Following is the set of local predicates that each node has to compute:

#### 3.2.1 Predicates for the root node $r$

1. $\text{parent}_r = \text{null}$.
2. $\text{anc}(r) = \phi$.

#### 3.2.2 Predicates for a non-root node $v$

1. $\text{parent}_v \neq \text{null}$.
2. $\text{anc}(v) \neq \phi$.
3. $\text{parent}_v = \text{parent}_l(v)$. The parent of $v$ on $T$ denoted by $\text{parent}_v$ is same as $v$’s parent as computed by $v$ from the labels of $v$ and its neighbors.
4. $\text{interval}_v \subset \text{interval}_{\text{parent}_v}$.
5. $\forall u \in \text{anc}(v)$ such that $u \neq \text{parent}_v$ ($\text{interval}_{\text{parent}_v} \subset \text{interval}_u$).
3.2.3 Predicates for every node (a root as well as a non-root) \( v \)

1. \( \text{out}_v > \text{in}_v \).

2. There is no neighbor of \( v \) such that its interval is totally disjoint with \( v \). Formally
   \[ \forall u \in \eta(v) \ (\text{interval}_u \subset \text{interval}_v \vee \text{interval}_u \supset \text{interval}_v) \].

3. if \( |\text{children}_l(v)| = 0 \) then \( \text{out}_v = \text{in}_v + 1 \).

4. if \( |\text{children}_l(v)| > 0 \) and let \( \text{children}_D_l(v) \) denote the list of children of \( v \) sorted in ascending order of their \( \text{in} \) labels and \( \text{firstChild}_l(v) \) and \( \text{lastChild}_l(v) \) be the first and last members of \( \text{children}_D_l(v) \) then \( \text{in}_{\text{firstChild}_l} = \text{in}_v + 1 \wedge \text{out}_v = \text{out}_{\text{lastChild}_l} + 1 \).

5. if \( |\text{children}_l(v)| > 1 \) and let \( \text{children}_P_l(v) \) denote the list of children of \( v \) sorted in the ascending order of their port numbers in \( v \) or \( \alpha_v \), then \( \text{children}_D_l(v) \) and \( \text{children}_P_l(v) \) sort the members of \( \text{children}_l(v) \) in the same order.

6. Let \( u \) and \( w \in \text{desc}_l(v) \), \( u \neq w \), such that \( u \in \text{children}_l(v) \) and \( w \notin \text{children}_l(v) \) then \( \alpha_v(u) < \alpha_v(w) \).

7. \( \forall (u, w) \in \text{children}_D_l(v) \) such that \( u \) and \( w \) are adjacent in \( \text{children}_D_l(v) \) and \( \text{in}_u < \text{in}_w \), then \( \text{in}_u = \text{out}_u + 1 \)

4 Generating the Labels: Marker \( \mathcal{M} \)

A natural method for assigning the \((\text{in}, \text{out})\) labels is to perform an actual DFS traversal of the network starting from the root. The required labels can be generated by augmenting some known DFS tree construction algorithm (e.g. [3], [2], [6]) by adding new variables for the labels and specific actions for updating these label variables. Algorithm 4.1 is such an adaptation of a DFS Tree construction algorithm. This algorithm acts as the marker \( \mathcal{M} \) of the proof labeling scheme. Note that \( \mathcal{M} \) is not self-stabilizing.

5 The Silent-Stabilizing DFS Construction Algorithm

We have constructed a proof labeling scheme \((\mathcal{M}, \mathcal{V})\) with a non-stabilizing marker \( \mathcal{M} \) that takes as input a connected graph \( G \) and assigns \((\text{in}, \text{Out})\) labels to every node in \( G \). It also has a verifier \( \mathcal{V} \) that takes as input a labeled (with \((\text{in}, \text{out})\) intervals) distributed data structure and verifies whether the input structure is a DFS tree. The proofs for the correctness and the performance of \((\mathcal{M}, \mathcal{V})\) are presented in Section 6. In the meanwhile, we use them here assuming they are correct.
| Variable     | Description                                                                 |
|--------------|-----------------------------------------------------------------------------|
| $N_v$        | The set of neighbors of node $v$ (input to the algorithm).                 |
| $\text{parent}_v$ | The parent of node $v$ in the DFS tree (output of the algorithm).       |
| $\text{in}_v$ | The $\text{in}$ label of node $v$ (output of the algorithm).              |
| $\text{out}_v$ | The $\text{out}$ label of node $v$ (output of the algorithm).            |
| $\text{exploreStatus}_v$ | The current exploration status of node $v$, it can be one of \{idle, done\} $\cup N_v$. |
| $\text{currentDFS}_v$ | The current DFS number being explored by node $v$.                     |

**Initializations at node $v$:**
1. $\text{exploreStatus} \leftarrow \text{idle}$

**Routine for finding the next child of $v$:**
2. $\text{findNextChild}(v) \equiv (u = \min_{u' \in N_v} \{u' : (\alpha_v(u') > \alpha_v(\text{exploreStatus}_v) \land (\text{exploreStatus}_{u'} = \text{idle}))\})$ if $u$ exists, $\text{done}$ otherwise

**Routine for finding the parent of $v$:**
3. $\text{findParent}(v) \equiv u \in N_v :: \text{exploreStatus}_u = v$

**Routine for finding the returning neighbor of $v$:**
4. $\text{findReturningChild}(v) \equiv u \in N_v :: \text{exploreStatus}_v = u \land \text{exploreStatus}_u = \text{done}$

**Predicates for computing the guard conditions of actions at $v$**
5. $\text{Forward}(v) \equiv \text{exploreStatus}_v = \text{idle}$ if $v$ is the root, $\text{exploreStatus}_v = \text{idle} \land \exists u \in N_v :: \text{exploreStatus}_u = v$ otherwise
6. $\text{Backward}(v) \equiv \text{exploreStatus}_v = u \land \text{exploreStatus}_u = \text{done}$  
   /*Actions executed by root when its $\text{Forward}(v)$ predicate evaluates to true*/
7. $\text{Forward(root)} \rightarrow \text{parent}_\text{root} = \text{null}; \text{exploreStatus}_\text{root} = \text{findNextChild(root)}; \text{In}_\text{root} = 0; \text{CurrentDFS}_\text{root} = 1;  
   /*Actions executed by $v :: v \neq \text{root}$ when its $\text{Forward}(v)$ predicate evaluates to true*/
8. $\text{Forward}(v) \rightarrow$
9. $\text{parent}_v = \text{findParent}(v); \text{exploreStatus}_v = \text{findNextChild}(v); \text{In}_v = \text{currentDFS}_{\text{findParent}(v)}; \text{currentDFS}_v = \text{currentDFS}_{\text{findParent}(v)} + 1;$  
   if($\text{findNextChild}_v = \text{done} \text{Out}_v = \text{CurrentDFS}_{\text{findParent}(v)} + 1$;  
   /*Actions executed by node $v$, root as well as non root when its $\text{Backward}(v)$ predicate evaluates to true*/
10. $\text{Backward}(v) \rightarrow \text{exploreStatus}_v = \text{findNextChild}(v); \text{currentDFS}_v = \text{CurrentDFS}_{\text{findReturningChild}(v)} + 1; \text{if}(\text{findNextChild}(v) = \text{done}) \text{out}_v = \text{CurrentDFS}_{\text{findReturningChild}(v)} + 1$;

**Algorithm 4.1**: Marker Algorithm
A simple way to stabilize any input/output algorithm is to run the algorithm repeatedly to maintain the correct output along with a self-stabilizing synchronizer [4]. This however would not be a silent algorithm. Still, let us use this approach to generate a non-silent self-stabilizing algorithm as an exercise, before presenting the silent one. Awerbuch and Varghese, in their seminal paper [4], present a transformer algorithm for converting a non-stabilizing input/output algorithm into its self-stabilizing version. Following theorem is taken from the seminal paper of Awerbuch and Varghese [4]:

**Theorem 1.** Given a non-stabilizing distributed algorithm \( \Pi \) to compute an input/output relation with a space complexity of \( S_\Pi \) and a time complexity of \( T_\Pi \). The Resynchronizer compiler produces a self-stabilizing version of \( \Pi \) whose time complexity is \( O(T_\Pi + \hat{D}) \) and whose space complexity is same as that of \( \Pi \).

Informally, the transformer that Awerbuch and Varghese developed to prove the above theorem is a self-stabilizing synchronizer. The transformer takes as input a non-stabilizing input/output algorithm \( \Pi \) whose running time and space requirement are \( T_\Pi \) and \( S_\Pi \) respectively. Another input it takes is \( \hat{D} \) which is an upper bound on the actual diameter \( D \) of the network. Given these inputs, the transformer performs \( \Pi \) for \( T_\Pi \) (recall that the transformer is a synchronizer and transforms the network to be synchronous). Then the transformer retains the results, performs \( \Pi \) again and compares the new results to the old ones. If they are the same, the old results are retained. if they differ, then some faults occurred, the new results are retained. This is repeated forever.

Since we do not assume the knowledge of \( n \) (required for input : \( T_M \)) or \( \hat{D} \), we use a slightly modified version of theorem [4] here, that appeared in [20]. The modified Awerbuch Varghese theorem presented in [20] is as follows:

**Theorem 2.** Given a non-stabilizing distributed algorithm \( \Pi \) to compute an input/output relation with a space complexity of \( S_\Pi \) and a time complexity of \( T_\Pi \). The enhanced Resynchronizer compiler produces a self-stabilizing version of \( \Pi \) whose time complexity is \( O(T_\Pi + n + D) \) and whose space complexity is \( O(S_\Pi + \log n) \).

Informally, Korman et al used a better synchronizer plus a simple self-stabilizing algorithm that computes \( n \) and \( D \) to prove the above theorem. To obtain a non-silent self-stabilizing DFS construction algorithm, we just plug the Marker \( M \) of Section 4 into theorem 2 and obtain the following corollary.

**Corollary 1.** There exists a non-silent self-stabilizing DFS construction algorithm that can operate in a dynamic asynchronous network, with a time complexity of \( O(T_M + n + D) \) and a space complexity of \( O(S_M + \log n) \).

### 5.1 Achieving Silent-Stabilization

The self-stabilizing DFS construction algorithm obtained above is not silent since the marker \( M \) is running forever and so is the Resynchronizer. To make
the algorithm silent-stabilizing we run the Marker (along with the Resynchronizer) only once in the beginning to generate the labels. Once the Marker finishes constructing the DFS tree and generating the labels, we switch the Resynchronizer off. From here on, the nodes start running the Verifier$V$. Since$V$can detect a fault in exactly one pulse if one occurs, we can manage without running the Resynchronizer during the verification. The verifier gives us a set of detecting nodes in case of a fault. If a fault occurs, the detecting nodes issue a reset. The reset restarts the Resynchronizer and once all the nodes regain synchronization (This is ensured by the reset protocol used in the Resynchronizer of [20]) they start constructing the DFS tree again by invoking the Marker. Thus we obtain a silent-stabilizing DFS construction algorithm. The following theorem summarizes our result:

**Theorem 3.** The proof labeling scheme$\langle M, V \rangle$for a DFS tree implies a silent-stabilizing DFS construction algorithm, that runs in$O(T_M + n + D)$time with a space complexity of$O(S_M + S_V + \log n)$.

### 6 Correctness and Performance Analysis

In this section, we establish the correctness of the proof labeling scheme$\langle M, V \rangle$described earlier. The proofs follow easily from the known properties of DFS tree and the predicates of the verifier. Given a labeled (with (in,out) labels) graph$G(V,E)$and the distributed representation of a spanning subgraph$T$of$G$, the following lemmas holds on$G$if the local interval predicates (Section 3.2) hold true at every node of$G$:

**Lemma 1.**$T$is a spanning tree of$G$.

**Proof.** In order to prove that a graph is a tree, it is sufficient to prove that it has no cycles and its number of edges is$n - 1$, where$n$is the number of nodes in this graph [15]. For the subgraph$T$of$G$to have a cycle, one of the ancestors of some node$v ∈ V$has to mark$v$as its parent. But this is ruled out by the predicate 4 of Section: 3.2.2 which requires that the interval of a node be included in the interval of its parent. Inductively applying the predicate 4 of Section: 3.2.2 to$v$and its ancestors, it follows that the interval of an ancestor of$v$includes$v$'s interval, therefore it is not possible for an ancestor of$v$to pick$v$as its parent without violating the predicate 4. The parent pointer of each node$v ∈ V$except the root comprises of a single incident edge of$v$and the parent pointer of the root is null, therefore there are exactly$n$nodes and$n - 1$edges in$T$.  

**Observation 1.** The macros defined in Section 3.1 extract (periodically) a perceived tree$T_l$from the (in,out) labels of the nodes in$G$.

While input tree$T$is encoded only by the collection of the parent pointers of the nodes,$T_l$is extracted by having each node compute its perceived parent as well as its perceived children on$T_l$.  

11
Lemma 2. For any node \( v \in V \), the set of children of \( v \) in \( T \) is same as the set of perceived children on \( v \) on \( T_1 \): \( \text{children}_1(v) \).

Proof. In what follows, we prove that if a node \( v \) has a node \( p \) as its perceived parent (\( \text{parent}_1(v) \)), then \( v \in \text{children}_1(p) \) i.e. \( v \) belongs to the set of perceived children of \( p \). In other words, in \( T_1 \), the collection of perceived parents is consistent with the collection of perceived children. Along with predicate 3 of section 3.2.2 this proves the lemma.

Assume, for contradiction, that though \( v \) has \( p \) as parent \( \text{parent}_1(v) \), \( v \notin \text{children}_1(p) \). Note that, by definition of a perceived parent and simple inductive arguments, \( p \) has the ‘narrowest’ interval of any node whose interval contains \( \text{interval}_v \) i.e. interval of \( p \) does not contain the interval of any other node whose interval contains \( \text{interval}_v \). Since \( v \) has \( p \) as its parent, clearly \( v \) and \( p \) are neighbors. if \( v \notin \text{children}_1(p) \), this implies that there is a node \( x \in \eta(p) \) with \( \text{interval}_x \supset \text{interval}_v \) and moreover \( \text{interval}_p \supset \text{interval}_x \). This implies that \( p \) can not be the parent of \( v \).

Following lemma 2, in the discussion that follows, \( \text{children}_1(v) \) implies the children of \( v \) in \( T \) and vice versa.

Lemma 3. For any two children \( u,w \) of a node \( v \) in \( T \), the intervals of all the nodes in the subtree of \( u \) in \( T \) are disjoint from the intervals of all the nodes in the subtree of \( w \) in \( T \).

Proof. Recall that the set, \( \text{children}_D_1(v) \), is the set \( \text{children}_1(v) \) sorted in the ascending order of the in labels of the nodes \( \in \text{children}_1(v) \). Let us assume, without loss of generality, that \( \text{in}_w > \text{in}_u \). Let there be a node \( u' \in \eta(v) \) such that \( u' \) is adjacent to \( u \) and appears after \( u \) in \( \text{children}_D_1(v) \). It is possible that \( u' = w \). Applying predicate 7 of Section 3.2.3 to \( u \) and \( u' \), \( \text{in}_{u'} = \text{out}_u + 1 \). By predicate 1 of Section 3.2.3 \( \text{out}_{u'} > \text{in}_{u'} \). Both in and out labels of \( u' \) are greater than the the out label of \( u \). Applying predicate 4 of Section 3.2.2 inductively, it is easy to see that the intervals of all the descendants of \( u \) are included in its own interval. Similarly the intervals of all the descendants of \( u' \) are included in the interval of \( u' \). Therefore intervals of all the descendants of \( u \) are disjoint from the intervals of \( u' \) and all its descendants. By repeatedly applying the above argument to every adjacent pair of nodes in \( \text{children}_D_1(v) \) starting from \( u \) to \( w \), it is easy to show that the subtrees of any two children of a node have disjoint intervals.

Lemma 4. For any two children \( u,w \) of some node \( v \) in \( T \), every simple path in \( G \) from some node in the subtree of \( u \) to any node in the subtree of \( w \) in \( T \) goes through either \( v \) or \( v \)’s ancestors.

Proof. Let \( u' \) be some node in the subtree of \( u \) and \( w' \) be some node in the subtree of \( w \). Let us assume, by way of contradiction, that there is a simple path \( P \) in \( G \) between \( u' \) and \( w' \) that does not go through \( v \) or \( v \)'s ancestors. For \( P \) to bypass \( v \) and its ancestors, it must go through:

- either, the subtree of a sibling of \( u \) (including \( w \)).
• or, the subtree of a sibling of a common ancestor (including $v$) of $u$ and $w$.

Both these cases require an edge to exist in $G$ that connects a pair of nodes in two sibling subtrees, known as a cross edge. By lemma 3, the intervals of all the nodes in the subtree of some node $x$ are disjoint from the intervals of all the nodes in the subtree of a sibling of $x$. Thus, the existence of any such edge $G$ is ruled out by predicate 2 of Section 3.2.3.

**Theorem 4.** If a graph $G(V, E)$ has every node $v \in V$ labeled with its (in, out) interval and interval assignments are such that all the local interval predicates (Section 3.2) hold true at every node, then the spanning tree $T$ encoded in a distributed manner in the states of all the nodes of $G$ is the First DFS tree of $G$.

**Proof.** The problem of finding the first DFS Tree of a graph can be thought of as the one of selecting the lexicographically smallest simple path of every node $v \in V$ out of all the simple paths from the root to $v$, see [7]. Let $P^T_v$ denote the path leading from the root to some node $v$ in $T$. We now prove that for any node $v \in V$, $P^T_v$ is lexicographically smallest among all the simple paths from the root to $v$ in $G$. By way of contradiction, let us suppose that $P^T_v$ is not lexicographically smallest which implies that there is another simple path $P^\text{Alt}_v$ from the root to $v$ which is smaller than $P^T_v$. Let us assume, w.l.o.g., that $P^T_v$ and $P^\text{Alt}_v$ are the same up-to some node $v_m$, the $m$th node of the common prefix. Let $v^T_{m+1}$ and $v^\text{Alt}_{m+1}$ denote the $(m+1)^{th}$ node of $P^T_v$ and $P^\text{Alt}_v$ respectively. as $v^\text{Alt}_{m+1}$.

**Observation 2.** For $P^\text{Alt}_v$ to be lexicographically smaller than $P^T_v$, the edge
index (as defined in : Section 2) \( \alpha_v(m) \) must be smaller than the corresponding index \( \alpha_v(m') \).

There are only three possibilities for \( P^v \) based on how \( v_{m+1} \) is related to \( v_m \):

1. \( v_{m+1} \) is an ancestor of \( v_m \): This case is ruled out since any such path will not be a simple path.

2. \( v_{m+1} \) is a child of \( v_m \): \( v_{m+1} \) and \( v_T \) are both children of \( v_m \). According to lemma 4, there is no simple path from \( v_{m+1} \) to any node in the subtree of \( v_T \) that does not go through \( v_m \) or any of its ancestors. Since \( v_T \) falls on \( P^v \), \( v \) belongs to the subtree of \( v_T \) in \( T \). Thus, there is no simple path connecting \( v_{m+1} \) to \( v \) that does not go through \( v_m \) or its ancestors. The path from \( v_{m+1} \) to \( v \) that goes through either \( v_m \) or any of its ancestors would not be a simple path as in case 1. Therefore, this case is also ruled out.

3. \( v_{m+1} \) is a proper descendant (a descendant which is not a child) of \( v_m \): This case can be further subdivided into two sub cases:

   (a) \( v_{m+1} \) is also a descendant of \( V_T \) in addition to being a descendant of \( v_m \): \( v_T \) is a child of \( v_m \), whereas \( v_{m+1} \) is a proper descendant of \( v_m \). This implies that \( in_{m+1} > in_T \). This is in contradiction to local interval predicate 6 (Section 3.2.3) which requires that the edge index of the edge \( (v_m, v_{m+1}) \) be smaller than the edge index of the edge \( (v_m, v_T) \) in \( \alpha v_m \).

   (b) \( v_{m+1} \) is a proper descendant of \( v_m \), but not a descendant of \( v_T \): This case is similar to that of 2.

\[ \square \]

**Theorem 5.** The verifier \( V \) described in section \( 3 \) runs in time \( O(1) \) and requires \( O(\log n) \) bits of memory per node.

**Proof.** The running time of \( V \) follows from the fact that each node needs to look only at the labels of its immediate neighbors in order to compute its predicates. Every node shares its \( (\text{in, out}) \) labels with its neighbors. The maximum value of a label is \( 2^n \) which can be encoded using \( O(\log n) \) bits.

The following theorem establishes the correctness and performance of the marker \( M \):

**Theorem 6.** Algorithm 4.1 constructs the First DFS tree and assigns \( (\text{in, out}) \) labels to all the nodes of the input graph \( G(V, E) \) in time \( O(n) \) using \( O(\log n) \) bits of memory per node.
Proof. It is easy to see that algorithm 4.1 adds new actions to a standard DFS tree construction algorithm for updating the in and out labels. These actions do not change the values of any of the variables of the original algorithm. Also, we do not change the algorithms flow of control, except for adding these actions. Since each action is just a new assignment to a new variable (of logarithmic size), the addition of these actions cannot violate the correctness of the construction algorithm, nor change its time complexity.

A node $v$ always picks the unvisited neighbor with the smallest port number (line 2 of algorithm 4.1) as the NextChild$_v$. This ensures that the output of the algorithm is the First DFS tree of the input graph. The standard DFS tree construction algorithm without any actions for updating the (in, out) labels has a space complexity of $O(\log \Delta)$ bits per node for encoding the $(\Delta + 2)$ possible values of State$_v$. The variable CurrentDFS$_v$ in the algorithm 4.1 used for updating the (in, out) labels requires $\log n$ bits per node. Therefore the overall space complexity of $M$ is $O(\log n)$.

Plugging in the time and space complexities of $M$ and $V$ in theorem 3, we get the following corollary:

**Corollary 2.** The proof labeling scheme $(M, V)$ for DFS tree implies a silent-stabilizing DFS construction algorithm, that runs in $O(n)$ time with a space complexity of $O(\log n)$.

### 7 Self-stabilizing DFS token circulation

The silent-stabilizing DFS tree of Section 5.1 can be combined with a self-stabilizing mutual exclusion algorithm for tree networks to obtain a self-stabilizing token circulation scheme for general networks with a specified root. Self-stabilizing mutual exclusion algorithms that circulate a token in the DFS order on a tree network can be found in [14, 22, 25]. Petit and Villain presented a space optimal snap-stabilizing DFS token circulation algorithm for tree networks in [27] with a waiting time of $O(n)$. We can combine our silent-stabilizing DFS tree with the snap stabilizing DFS token circulation protocol of [27] using the fair composition method to obtain a DFS token circulation for general networks. The space complexity of [27] is $O(\log \Delta)$ and that of our silent-stabilizing DFS tree is $O(\log n)$. Therefore the space complexity of the resulting self-stabilizing DFS token circulation algorithm is $O(\log n)$.

**Acknowledgements**

We would like to thank Amitabh Trehan for useful comments and discussions.

---

$^1$ Actually, these actions are the distributed version of common actions of various versions of non-distributed DFS.
References

[1] Yehuda Afek, Shay Kutten, and Moti Yung. The local detection paradigm and its applications to self-stabilization.

[2] Baruch Awerbuch. A new distributed depth-first-search algorithm. Information Processing Letters, 20(3):147–150, 1985.

[3] Baruch Awerbuch, Boaz Patt-Shamir, George Varghese, and Shlomi Dolev. Self-stabilization by local checking and global reset (extended abstract). In Proceedings of the 8th International Workshop on Distributed Algorithms, WDAG ’94, pages 326–339, London, UK, UK, 1994. Springer-Verlag.

[4] Baruch Awerbuch and George Varghese. Distributed program checking: A paradigm for building self-stabilizing distributed protocols (extended abstract). In Proceedings of the 32Nd Annual Symposium on Foundations of Computer Science, SFCS ’91, pages 258–267, Washington, DC, USA, 1991. IEEE Computer Society.

[5] Imrich Chlamtac and Shay Kutten. Tree-based broadcasting in multihop radio networks. Computers, IEEE Transactions on, 100(10):1209–1223, 1987.

[6] Isreal Cidon. Yet another distributed depth-first-search algorithm. Inf. Process. Lett., 26(6):301–305, January 1988.

[7] Zeev Collin and Shlomi Dolev. Self-stabilizing depth-first search. Information Processing Letters, 49(6):297 – 301, 1994.

[8] Alain Cournier, Stephane Devismes, Franck Petit, and Vincent Villain. Snap-stabilizing depth-first search on arbitrary networks. Comput. J, pages 268–280, 2006.

[9] Alain Cournier, Stphane Devismes, and Vincent Villain. A snap-stabilizing dfs with a lower space requirement. In In Seventh International Symposium on Self-Stabilizing Systems (SSS?05, pages 33–47, 2005.

[10] Ajoy K. Datta, Colette Johnen, Franck Petit, and Vincent Villain. Self-stabilizing depth-first token circulation in arbitrary rooted networks. Distrib. Comput., 13(4):207–218, November 2000.

[11] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. Commun. ACM, 17(11):643–644, November 1974.

[12] Shlomi Dolev. Self-stabilization. MIT Press, 2000.

[13] Shlomi Dolev, Mohamed G. Gouda, and Marco Schneider. Memory requirements for silent stabilization. Acta Informatica, 36(6):447–462, 1999.
[14] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distrib. Comput.*, 7(1):3–16, November 1993.

[15] Shimon Even. *Graph Algorithms*. W. H. Freeman & Co., New York, NY, USA, 1979.

[16] Shing-Tsaan Huang and Nian-Shing Chen. Self-stabilizing depth-first token circulation on networks. *Distributed Computing*, 7(1):61–66, 1993.

[17] Colette Johnen, Gianluigi Alari, Joffroy Beauquier, and Ajoy K Datta. Self-stabilizing depth-first token passing on rooted networks. In *Distributed Algorithms*, pages 260–274. Springer, 1997.

[18] Colette Johnen and Joffroy Beauquier. Space-efficient, distributed and self-stabilizing depth-first token circulation. In *Proceedings of the Second Workshop on Self-Stabilizing Systems*, pages 4–1, 1995.

[19] Shmuel Katz and Kenneth J. Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7(1):17–26, 1993.

[20] Amos Korman and Shay Kutten. Distributed verification of minimum spanning trees. In *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, PODC ’06, pages 26–34, New York, NY, USA, 2006. ACM.

[21] Amos Korman, Shay Kutten, and David Peleg. Proof labeling schemes. *Distributed Computing*, 22(4):215–233, 2010.

[22] Franck Petit. Highly space-efficient self-stabilizing depth-first token circulation for trees. In *Euro-par’97 Parallel Processing, Proceedings LNCS*, pages 476–479. Springer-Verlag, 1997.

[23] Franck Petit. Fast self-stabilizing depth-first token circulation. In *Proceedings of the 5th International Workshop on Self-Stabilizing Systems, WSS ’01*, pages 200–215, London, UK, UK, 2001. Springer-Verlag.

[24] Franck Petit and Vincent Villain. Color optimal self-stabilizing depth-first token circulation. In *ISPAN*, pages 317–323. IEEE Computer Society, 1997.

[25] FRANCK PETIT and VINCENT VILLAIN. Optimality and self-stabilization in rooted tree networks. *Parallel Processing Letters*, 10(01):3–14, 2000.

[26] Franck Petit and Vincent Villain. Self-stabilizing depth-first token circulation in asynchronous message-passing systems. *Computers and Artificial Intelligence*, 19(5), 2000.

[27] Franck Petit and Vincent Villain. Optimal snap-stabilizing depth-first token circulation in tree networks. *Journal of Parallel and Distributed Computing*, 67(1):1 – 12, 2007.
[28] George Varghese and Mahesh Jayaram. The fault span of crash failures. *Journal of the ACM, 47:47–2*, 2000.