Snap-Stabilizing Depth-First Search on Arbitrary Networks[⋆]

Alain Cournier, Stéphane Devismes, Franck Petit, and Vincent Villain

LaRIA, CNRS FRE 2733,
Université de Picardie Jules Verne, Amiens (France)

Abstract. A *snap-stabilizing protocol*, starting from any arbitrary initial configuration, always behaves according to its specification. In this paper, we present a snap-stabilizing depth-first search wave protocol for arbitrary rooted networks. In this protocol, a wave of computation is initiated by the root. In this wave, all the processors are sequentially visited in depth-first search order. After the end of the visit, the root eventually detects the termination of the process. Furthermore, our protocol is proven assuming an unfair daemon, i.e., assuming the weakest scheduling assumption.

keywords: Distributed systems, fault-tolerance, stabilization, depth-first search.

1 Introduction

A distributed system is a network where processors execute local computations according to their state and the messages from their neighbors. In such systems, a *wave protocol* [1] is a protocol where at least one processor (called *initiator*) initiates cycles of computations (also called *wave*). At the ending of each cycle, each initiator is abled to determine a result depending on both the terminal configuration and the history of the cycle's computation.

In an arbitrary rooted network, a Depth-First Search (DFS) wave is initiated by the root. In this wave, all the processors are sequentially visited in depth-first search order. This scheme has many applications in distributed systems. For example, the solution of this problem can be used to solve mutual exclusion, spanning tree computation, constraint programming, routing, or synchronization.

The concept of *$self$-stabilization* [2] is the most general technique to design a system to tolerate arbitrary transient faults. A self-stabilizing system, regarless of the initial states of the processors and messages initialy in the links, is guaranteed to converge to the intented behavior in finite time. *Snap-stabilization* was introduced in [3]. A *snap-stabilizing* protocol guaranteed that it always behaves according to its specification. In other words, a snap-stabilizing protocol is also a

[⋆] The full version is available at www.laria.u-picardie.fr/∼devismes/tr2004-9.ps self-stabilizing protocol which stabilizes in 0 step. Obviously, a *snap-stabilizing* protocol is optimal in stabilization time.

Related Works. There exists several (non self-stabilizing) distributed algorithms solving this problem, e.g., [4, 5]. In the area of self-stabilizing systems, a silent algorithm (i.e., using this algorithm, the system converges to a fix point) which computes a DFS spanning tree for arbitrary rooted networks is given in [6]. Several self-stabilizing (but not snap-stabilizing) wave algorithms based on the $depth\text{-}first$ *token circulation* ($DFTC$) have been proposed for arbitrary rooted networks. The first one was proposed in [7]. It requires $O(\log(N) + \log(\Delta))$ bits per processors where N is the number of processors and Δ the degree of the network. Subsequently, several other self-stabilizing protocols were proposed, e.g., [8, 9, 10]. All these papers attempted to reduce the memory requirement to $O(\log(\Delta))$ bits per processor. The algorithm proposed in [8] offers the best space complexity. All these above solutions [8, 7, 9, 10] have a stabilization time in $O(N \times D)$ rounds where D is the diameter of the network. The solution proposed in [11] stabilizes in $O(N)$ rounds using $O(\log(N) + \log(\Delta))$ bits per processor. Until now, this is the best solution (for arbitrary networks) in term of trade-off between time and space complexities. The correctness of the above algorithms is proven assuming a (weakly) fair daemon. Roughly speaking, a daemon is considered as an adversary which tries to prevent the protocol to behave as expected, and fairness means that the daemon cannot prevent forever a processor to execute an enabled action. The first snap-stabilizing $DFTC$ has been proposed in [12] for tree networks. In arbitrary networks, a *universal transformer* providing a snap-stabilizing version of any (neither self- nor snap-) protocol is given in [13]. Obviously, combining this protocol with any $DFTC$ algorithm, we obtain a snap-stabilizing $DFTC$ algorithm for arbitrary networks. However, the resulting protocol works assuming a weakly fair daemon only. Indeed, it generates an infinite number of snapshots, independently of the token progress. Therefore, the number of steps per wave cannot be bounded.

Contributions. In this paper, we present the first snap-stabilizing depth-first search wave protocol for arbitrary rooted networks assuming an unfair daemon, i.e., assuming the weakest scheduling assumption. Indeed, using our protocol, the execution of a DFS wave is bounded by $O(N^2)$ steps. The protocol does not use any pre-computed spanning tree but requires identities on processors. The snap-stabilizing property guarantees that as soon as the protocol is initiated by the root, every processor of the network will be visited in DFS order. After the end of the visit, the root eventually detects the termination of the process.

Outline of the paper. The rest of the paper is organized as follows: in Section 2, we describe the distributed systems and the model in which our protocol is written. Moreover, in the same section, we give a formal statement of the Depth-First Search Wave Protocol solved in this paper. In Section 3, we present the Depth-First Search Wave Protocol. In the following section (Section 4), we give the proof of snap-stabilization of the protocol and some complexity results. Finally, we make concluding remarks in Section 5.

2 Preliminaries

Distributed System. We consider a *distributed system* as an undirected connected graph $G = (V, E)$ where V is a set of *processors* ($|V| = N$) and E is the set of *bidirectional communication links*. We consider networks which are *asynchronous* and *rooted*, i.e., among the processors, we distinguish a particular processor called *root*. We denote the root processor by r. A communication link (p, q) exists if and only if p and q are neighbors. 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 $Neig_p$[1], are locally ordered by $\prec_p$. We assume that $Neig_p$ is a constant, $Neig_p$ is shown as an input from the system. Moreover, we assume that the network is identified, i.e., every processor has exactly one identity which is unique in the network. We denote the identity of a processor p by Id_p. We assume that Id_p is a constant. Id_p is also shown as an input from the system.

Computational Model. In the computation model that we use, each processor executes the same program except r. We consider the local shared memory model of communication. The program of every processor consists of a set of *shared variables* (henceforth, referred to as variables) and a finite set of actions. A processor can only write to its own variables, and read its own variables and variables owned by the neighboring processors. 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 the 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. We assume that the actions are atomically executed, meaning, the evaluation of a guard and the execution of the corresponding statement of an action, if executed, are done in one atomic step.

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 ($\in V$). We will refer to the state of a processor and system as a (*local*) *state* and (*global*) *configuration*, respectively. Let $\mathcal{C}$, the set of all possible configurations of the system. An action A is said to be *enabled* in $\gamma \in \mathcal{C}$ at p if the guard of A is true at p in γ. A processor p is said to be *enabled* in γ ($\gamma \in \mathcal{C}$) if there exists an enabled action A in the program of p in γ.

Let a distributed protocol $\mathcal{P}$ be a collection of binary transition relations denoted by $\mapsto$, on $\mathcal{C}$. A *computation* of a protocol $\mathcal{P}$ is a *maximal* sequence of configurations $e = (\gamma_0,\gamma_1,...,\gamma_i,\gamma_{i+1},...)$, such that for $i \geq 0$, $\gamma_i \mapsto \gamma_{i+1}$ (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 $\mathcal{P}$ is enabled in the terminal configuration) or infinite. All computations considered in this paper are assumed to be maximal. The set of all possible computations of $\mathcal{P}$ is denoted as $\mathcal{E}$.

[1] Every variable or constant X of a processor p will be noted X_p.

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 action between these two configurations. (The disabling action represents the following situation: at least one neighbor of p changes its state between γ_i and γ_{i+1}, and this change effectively made the guard of all actions of p false.)

In a step of computation, first, all processors check the guards of their actions. Then, some *enabled* processors are chosen by a *daemon*. Finally, the "elected" processors execute one or more of theirs *enabled* actions. There exists several kinds of *daemon*. Here, we assume a *distributed daemon*, i.e., during a computation step, if one or more processors are enabled, the daemon chooses at least one (possibly more) of these enabled processors to execute an action. Furthermore, a daemon can be *weakly fair*, i.e., if a processor p is continuously enabled, p will be eventually chosen by the daemon to execute an action. If the daemon is *unfair*, it can forever prevent a processor to execute an action except if it is the only enabled processor.

In order to compute the time complexity, we use the definition of *round* [14]. 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 the disabling action) of every enabled processor from the first 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. We say that a round is *finite* if it is constituted of a finite number of steps.

Snap-stabilizing Systems. The concept of *snap-stabilization* was introduced in [3]. In this paper, we restrict this concept to the wave protocols only.

Definition 1 (Snap-stabilization for Wave Protocols). *Let $\mathcal{T}$ be a task, and $\mathcal{SP}_\mathcal{T}$ a specification of $\mathcal{T}$. A wave protocol $\mathcal{P}$ is snap-stabilizing for the specification $\mathcal{SP}_\mathcal{T}$ if and only if:*

1. *At least one processor (called initiator) eventually executes a particular action of $\mathcal{P}$ (called initialization action).*
2. *The result obtained with $\mathcal{P}$ from this initialization action always satisfies $\mathcal{SP}_\mathcal{T}$.*

Theorem 1. *Let $\mathcal{T}$ be a task and $\mathcal{SP}_\mathcal{T}$ be a specification of $\mathcal{T}$. Let $\mathcal{P}$ be a protocol such that, assuming a weakly fair daemon, $\mathcal{P}$ is self-stabilizing for $\mathcal{SP}_\mathcal{T}$. If, for every execution of $\mathcal{P}$ assuming an unfair daemon, each round is finite, then $\mathcal{P}$ is also self-stabilizing for $\mathcal{SP}_\mathcal{T}$ assuming an unfair daemon.*

Proof. Let e be an execution of $\mathcal{P}$ assuming an unfair daemon. By assumption, every round of e is finite. Then, as every round of e is finite, each enabled processor (in e) executes an action (either a disabling action or an action of $\mathcal{P}$) in a finite number of steps. In particular, every continuously enabled processor executes an action of $\mathcal{P}$ in a finite number of steps. So, e is also an execution of

$\mathcal{P}$ assuming a weakly fair daemon. Since $\mathcal{P}$ is self-stabilizing for $\mathcal{SP}_{\mathcal{T}}$ assuming a weakly fair daemon, e stabilizes to $\mathcal{SP}_{\mathcal{T}}$. Hence, $\mathcal{P}$ is self-stabilizing for $\mathcal{SP}_{\mathcal{T}}$ even if the daemon is unfair. $\square$

Specification of the Depth-First Search Wave Protocol. Before giving the specification of the Depth-First Search Wave Protocol, we propose some definitions.

Definition 2 (Path). *The sequence of processors p_1, ..., p_k ($\forall i \in [1... k]$, $p_i \in V$) is a path of $G = (V, E)$ if $\forall i \in [1...k-1]$, (p_i, p_{i+1}) $\in E$. The path p_1, ..., p_k is referred to as an elementary path if $\forall i$, j such that $1 \leq i < j \leq k$, $p_i \neq p_j$. The processors p_1 and p_k are termed as initial and final extremities, respectively.*

Definition 3 (First Path). *For each elementary path of G from the root, P = (p_1=r), ..., p_i, ...,p_k, we associate a word l_1, ..., l_i, l_{k-1} (noted word(P)) where, $\forall i \in [1...k-1]$, p_i is linked to p_{i+1} by the edge labelled l_i on p_i. Let $\prec_{lex}$ be a lexicographical order over these words. For each processor p, we define the set of all elementary paths from r to p. The path of this set with the minimal associated word by $\prec_{lex}$ is called the first path of p (noted $fp(p)$).*

Using this notion, we can define the *first DFS order*:

Definition 4 (First DFS Order). *Let p, $q \in V$ such that $p \neq q$. We can define the first DFS order $\prec_{dfs}$ as follows: $p \prec_{dfs} q$ if and only if $word(fp(p)) \prec_{lex} word(fp(q))$.*

Specification 1 (fDFS Wave). *Let $Visited$ be a predicate. A finite computation $e \in \mathcal{E}$ is called a fDFS wave (i.e.,* first DFS *wave) if and only if the following tree conditions are true:*

1. *r initiates the fDFS wave by initializing the set of processors satisfying Visited with r.*
2. *During a fDFS wave, the other processors are sequentially included in the set of processors satisfying Visited following the first DFS order.*
3. *r eventually detects the ending of a fDFS wave and if r detects the ending of a fDFS wave then $\forall p \in V$, p satisfies Visited.*

Remark 1. In order to prove that our protocol is snap-stabilizing for Specification 1, we must show that every execution of the protocol satisfies these both conditions:

1. r eventually initiates a fDFS wave.
2. From any configuration where r has initiated a fDFS wave, the system always satisfies Specification 1.

3 Algorithm

In this section, we present a *DFS* wave protocol referred to as Algorithm *snap$\mathcal{DFS}$* (see Algorithms 1. and 2.). We first present the normal behavior. We then explain the method of error correction.

Algorithm 1. Algorithm *snap$\mathcal{DFS}$* for $p = r$

```
Input: $Neig_p$: set of neighbors (locally ordered); $Id_p$: identity of $p$;
Constant: $Par_p = \perp$;
Variables: $S_p \in Neig_p \cup \{idle, done\}$; $Visited_p$: set of identities;
Macros:
$Next_p$ = $(q = \min_{\prec_p}\{q' \in Neig_p :: (Id_{q'} \notin Visited_p)\})$ if $q$ exists,
        $done$ otherwise;
$ChildVisited_p = Visited_{S_p}$ if $(S_p \notin \{idle, done\})$, $\emptyset$ otherwise;
Predicates:
$Forward(p)$ $\equiv (S_p = idle)$
$Backward(p)$ $\equiv (\exists q \in Neig_p :: (S_p = q) \wedge (Par_q = p) \wedge (S_q = done))$
$Clean(p)$ $\equiv (S_p = done)$
$SetError(p)$ $\equiv (S_p \neq idle) \wedge [(Id_p \notin Visited_p)$
        $\vee (\exists q \in Neig_p :: (S_p = q) \wedge (Id_q \in Visited_p))]$
$Error(p)$ $\equiv SetError(p)$
$ChildError(p) \equiv (\exists q \in Neig_p :: (S_p = q) \wedge (Par_q = p) \wedge (S_q \neq idle)$
        $\wedge \neg(Visited_p \subsetneq Visited_q))$
$LockedF(p)$ $\equiv (\exists q \in Neig_p :: (S_q \neq idle))$
$LockedB(p)$ $\equiv [\exists q \in Neig_p :: (Id_q \notin ChildVisited_p) \wedge (S_q \neq idle)] \vee Error(p)$
        $\vee ChildError(p)$
Actions:
$F :: Forward(p) \wedge \neg LockedF(p)$ $\rightarrow Visited_p := \{Id_p\}$; $S_p := Next_p$;
$B :: Backward(p) \wedge \neg LockedB(p)$ $\rightarrow Visited_p := ChildVisited_p$; $S_p := Next_p$;
$C :: Clean(p) \vee Error(p)$ $\rightarrow S_p := idle$;
```

Normal Behavior. From a normal configuration, we distinguish two phases in our protocol: the *visiting phase* where the protocol visits all the processors in the *first DFS order* and the *cleaning phase* which cleans the trace of the *visiting phase* so that the root is eventually ready to initiate a new *visiting phase.* These both phases work in parallel. In its normal behavior, Algorithm *snap$\mathcal{DFS}$* uses three variables for each processor p:

1. S_p designates the *successor* of p in the visiting phase, i.e., if there exists $q \in Neig_p$ such that $S_p = q$, then q (resp. p) is said to be a *successor* of p (resp. a *predecessor* of q),
2. $Visited_p$ is the set of processors which have been visited during the visiting phase,
3. Par_p designates the processor which has pointed out p as one of its successors during the visiting phase (as r has no predecessor, Par_r is the constant $\perp$).

Consider the configurations where $[(S_r = idle) \wedge (\forall p \in Neig_r\ , S_p = idle) \wedge (\forall q \in V \setminus (Neig_r \cup \{r\}), S_q \in \{idle, done\})]$. We refer to these configurations as *normal initial configurations.* In these configurations, every processor $q \neq r$ such that $S_q = done$ is enabled to perform its cleaning phase (see Predicate $Clean(p)$). Processor q performs its cleaning phase by executing Action C, i.e., it assigns $idle$ to S_q. Moreover, in this configuration, the root (r) is enabled to initiate a visiting phase (Action F). Processor r can initiate a visiting phase by initializing $Visited_r$ with its identity (Id_r) and pointing out (with S_r) its minimal neighbor in the local order $\prec_r$ (see Macro $Next_r$). In the worst case, every processor q, such that $S_q = done$, executes its cleaning phase, after, r is the only enabled processor and initiates a visiting phase. From this point on, r is the only *visited* processor.

When a processor $p \neq r$ such that $S_p = idle$ is pointed out with S_q by a neighboring processor q, then p waits until all its neighbors p', such that $S_{p'} = done$ and $Id_{p'} \notin PredVisited_p$ (here, $Visited_q$), execute their cleaning phase. After, p can execute Action F. Then, p also designates q with Par_p and assigns $PredVisited_p \cup \{Id_p\}$ (here, $Visited_q \cup \{Id_p\}$) to $Visited_p$. Informally, the $Visited$ set of the last visited processor contains the identities of all the visited processors. Finally, p chooses a new successor, if any. For this earlier task, two cases are possible (see Macro $Next_p$):

1. $\forall\, p' \in Neig_p$, $Id_{p'} \in Visited_p$, i.e., all neighbors of p have been visited; the visiting phase from p is now terminated, so, S_p is set to *done*,
2. otherwise, p chooses as a successor the minimal processor by $\prec_p$ in $\{p' :: p' \in Neig_p \wedge Id_{p'} \notin Visited_p\}$ and p is now in the visiting phase.

In both cases, p is now considered as *visited.*

When q is the successor of p and $S_q = done$, p knows that the visiting phase from q is terminated. Thus, p must continue the visiting phase using another neighboring processor which is still not visited, if any: p executes Action B and it assigns $ChildVisited_p$ to $Visited_p$. Hence, it knows exactly which processors have been visited and it can designate another successor, if any, as in Action F (see Macro $Next_p$). Processor q is, now, enabled to execute its cleaning phase (Action C).

Finally, $S_r = done$ means that the visiting phase is terminated for all the processors and so, r can execute its cleaning phase. Thus, the system eventually reaches a *normal initial configuration* again.

Error Correction. First, from the normal behavior, we can remark that, if $p \neq r$ is in the visiting phase and the visiting phase from p is still not terminated, then p must have a predecessor and must designate it with its variable Par_p, i.e., each processor $p \neq r$ must satisfy: $(S_p \notin \{idle,\, done\}) \Rightarrow (\exists q \in Neig_p :: S_q = p \wedge Par_p = q)$. The predicate $NoRealParent(p)$ allows to determine if this condition is not satisfied by p. Then, during the normal behavior, each processor maintains properties based on the value of its $Visited$ set and that of its predecessors, if any. Thus, in any configuration, p must respect the following conditions (see Action F):

1. $(S_p \neq idle) \Rightarrow (Id_p \in Visited_p)$ because when p is visited, it includes its identity in its $Visited$ set.
2. $(S_p \in Neig_p) \Rightarrow (Id_{S_p} \notin Visited_p)$, i.e., p must not point out a previously visited processor.
3. $((p \neq r) \wedge (S_p \neq idle) \wedge (\exists q \in Neig_p :: (S_q = p) \wedge (Par_p = q))) \Rightarrow (Visited_q \subsetneq Visited_p)$ because while $p \neq r$ is in the visiting phase, $Visited_p$ must strictly include the $Visited$ set of its parent.

Algorithm 2. Algorithm *snap$\mathcal{DFS}$* for $p \neq r$

```
Input:     $Neig_p$: set of neighbors (locally ordered); $Id_p$: identity of $p$;
Variables: $S_p \in Neig_p \cup \{idle, done\}$; $Visited_p$: set of identities; $Par_p \in Neig_p$;
Macros:
$Next_p$            $= (q = \min_{\prec_p}\{q' \in Neig_p :: (Id_{q'} \notin Visited_p)\})$ if $q$ exists,
                    $done$ otherwise;
$Pred_p$            $= \{q \in Neig_p :: (S_q = p)\}$;
$PredVisited_p$     $= Visited_q$ if $(\exists!\, q \in Neig_p :: (S_q = p))$, $\emptyset$ otherwise;
$ChildVisited_p$    $= Visited_{S_p}$ if $(S_p \notin \{idle, done\})$, $\emptyset$ otherwise;
Predicates:
$Forward(p)$        $\equiv (S_p = idle) \wedge (\exists q \in Neig_p :: (S_q = p))$
$Backward(p)$       $\equiv (\exists q \in Neig_p :: (S_p = q) \wedge (Par_q = p) \wedge (S_q = done))$
$Clean(p)$          $\equiv (S_p = done) \wedge (S_{Par_p} \neq p)$
$NoRealParent(p)$   $\equiv (S_p \notin \{idle, done\}) \wedge \neg(\exists q \in Neig_p :: (S_q = p) \wedge (Par_p = q))$
$SetError(p)$       $\equiv (S_p \neq idle) \wedge [(Id_p \notin Visited_p)$
                        $\vee (\exists q \in Neig_p :: (S_p = q) \wedge (Id_q \in Visited_p))$
                        $\vee (\exists q \in Neig_p :: (S_q = p) \wedge (Par_p = q) \wedge \neg(Visited_q \subsetneq Visited_p))]$
$Error(p)$          $\equiv NoRealParent(p) \vee SetError(p)$
$ChildError(p)$     $\equiv (\exists q \in Neig_p :: (S_p = q) \wedge (Par_q = p) \wedge (S_q \neq idle)$
                        $\wedge \neg(Visited_p \subsetneq Visited_q))$
$LockedF(p)$        $\equiv (|Pred_p| \neq 1) \vee (\exists q \in Neig_p :: (Id_q \notin PredVisited_p) \wedge (S_q \neq idle))$
                        $\vee (Id_p \in PredVisited_p)$
$LockedB(p)$        $\equiv (|Pred_p| \neq 1) \vee (\exists q \in Neig_p :: (Id_q \notin ChildVisited_p) \wedge (S_q \neq idle))$
                        $\vee Error(p) \vee ChildError(p)$
Actions:
$F :: Forward(p) \wedge \neg LockedF(p)$   $\rightarrow Visited_p := PredVisited_p \cup \{Id_p\}$;
                                          $S_p := Next_p$; $Par_p := (q \in Pred_p)$;
$B :: Backward(p) \wedge \neg LockedB(p)$  $\rightarrow Visited_p := ChildVisited_p$; $S_p := Next_p$;
$C :: Clean(p) \vee Error(p)$              $\rightarrow S_p := idle$;
```

If one of these conditions is not satisfied by p, p satisfies $SetError(p)$. So, Algorithm *snap$\mathcal{DFS}$* detects if p is in an abnormal state, i.e., $(((p \neq r) \wedge NoRealParent(p)) \vee SetError(p))$ with the predicate $Error(p)$. In the rest of the paper, we call *abnormal processor* a processor p satisfying $Error(p)$. If p is an abnormal processor, then we must correct p and all the processors visited from p. We simply correct p by setting S_p to *idle* (Action C). So, if, before p executes Action C, there exists a processor q such that $(S_p = q \wedge Par_q = p \wedge S_q \notin \{idle, done\} \wedge \neg Error(q))$, then after p executes Action C, q becomes an abnormal processor too (replacing p). These corrections are propagated until the visiting phase from p is completely corrected. However, during these corrections, the visiting phase from p can progress by the execution of Actions F and B. But, we can remark that the $Visited$ set of the last processor of a visiting phase grows by the execution of Actions F and B and the last processor of a visiting phase can only extend the propagation using processors which are not in its $Visited$ set. Thus, the visiting phase from an abnormal processor cannot run indefinitely. Hence, we will see later that the visiting phase from an abnormal processor will be eventually corrected.

Finally, we focus on the different ways to stop (or slow down) the propagation of the erroneous behaviors. Actions F and B allow a processor p to execute its visiting phase. However, by observing its state and that of its neighbors, p can detect some fuzzy behaviors and stop them: that is the goal of the predicates $LockedF(p)$ and $LockedB(p)$ in Actions F and B, respectively. A processor p is *locked* (i.e., p cannot execute Action B or Action F) when it satisfies at least one of the five following conditions:

1. p has several predecessors.
2. p is an abnormal processor.
3. p has a successor q such that $((S_q{\neq}idle){\wedge}(Par_q{=}p){\wedge}\neg(Visited_p{\subsetneq}Visited_q))$, i.e., q is abnormal.
4. p $(S_p = idle)$ is designated as a successor by q but Id_p is in $Visited_q$, i.e., q is abnormal.
5. some non-visited neighbors q of p are not cleaned, i.e., $S_q \neq idle$ (also used in a normal behavior).

4 Correctness and Complexity Analysis

4.1 Basic Definitions and Properties

Let $p \in V$. p is *pre-clean* if and only if $(Clean(p) \vee (S_p = done \wedge Error(p)))$. We recall that p is *abnormal* if and only if it satisfies $Error(p)$. A processor p is *linked* to a processor q if and only if $(S_p = q) \wedge (Par_q = p) \wedge \neg SetError(q) \wedge (S_q \neq idle)$. In this case p is called the *parent* of q and q the *child* of p. We can also remark that S_p (resp. Par_q) guarantees that q (resp. p) is the only child (resp. parent) of p (resp. q). As $Par_r = \perp$, obviously, r never has any parent.

A *linked path* of G is a path $P = p_1, ..., p_k$ such that $S_{p_1} \notin \{idle, done\}$ and $\forall$ i, $1 \leq i \leq k-1$, p_i is linked to p_{i+1}. We will note $IE(P)$ the *initial extremity* of P (i.e., p_1) and $FE(P)$ the *final extremity* of P (i.e., p_k). Moreover, the *length* of P (noted $length(P)$) is equal to k. Obviously, in any configuration, every linked path of G is elementary. So, from now on and until the end of the paper, we only consider maximal non-empty linked paths. The next lemma gives an important property of such linked paths.

Lemma 1. *Every linked path P satisfies $Visited_{FE(P)} \supseteq \{Id_p :: p \in P \wedge p \neq IE(P)\}$.*

We call abnormal linked path, a linked path P satisfying $Error(IE(P))$. Respectively, we call normal linked path, every linked path which is not abnormal. Obviously, a normal linked path P satisfies $IE(P) = r$.

Lemma 2. *A normal linked path P satisfies $Visited_{FE(P)} \supseteq \{Id_p :: p \in P\}$.*

Now, we introduce the notion of *future* of a linked path. We call *future* of a linked path P the evolution of P during a computation. In particular, the *immediate future* of P is the transformation supported by P after a step. Note that, after a step, P may disappear. Thus, by convention, we denote by $Dead_P$ the fact that P has disappeared after a step.

Definition 5 (Immediate Future of a Linked Path). *Let $\gamma_i \mapsto \gamma_{i+1}$ be a step. Let P be a linked path in γ_i. We call $F(P)$ the immediate future of P in γ_{i+1} and we define it as follows.*

1. **If** *there exists a linked path P' in γ_{i+1} which satisfies one of the following conditions: (a) $P \cap P' \neq \emptyset$, or (b) in γ_i, $S_{FE(P)} = IE(P')$ and $IE(P')$ executes Action F in $\gamma_i \mapsto \gamma_{i+1}$* **then** *$F(P) = P'$,*
2. **else**, *$F(P) = Dead_P$.*

By convention, we state $F(Dead_P) = Dead_P$.

Figure 1 depicts two types of immediate future. Consider first Configurations i and ii. Configuration i contains one linked path only: $P =$ r, 1, 2. Moreover, Processor 3 has Action F enabled in i and executes it in $i \mapsto ii$ (i.e., 3 hooks on to P). Thus, the step $i \mapsto ii$ illustrates the case 1.(a) of Definition 5: in this execution, $F(P) =$ r, 1, 2, 3. Configuration iii also contains one linked path only: $P' = 1$. Then, in iii, Processor 1 has Action C enabled and Processor 2 has Action F enabled. These two processors execute C and F respectively in $iii \mapsto iv$ (1 unhooks from P' and 2 hooks on to P'). So, we obtain Configuration iv which illustrates the case 1.(b) of Definition 5: in this execution, $F(P') = 2$. Note that if only Processor 1 executes Action C from iii, P' disappears, i.e., $F(P') = Dead_P$.

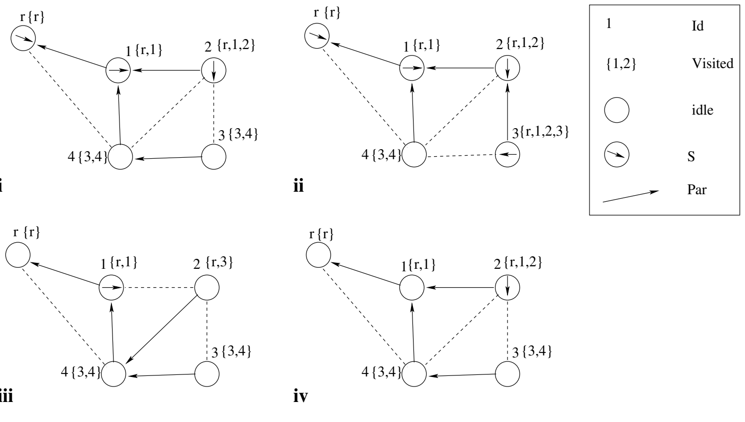

Fig. 1. Instances of Immediate Futures

Definition 6 (Future of a Linked Path). *Let $e \in \mathcal{E}$. Let $\gamma_i \in e$. We define $F^k(P)$ ($k \in \mathbb{N}$), the future of P in e after k steps of computation from γ_i, as follows:*

1. *$F^0(P) = P$,*
2. *$F^1(P) = F(P)$ (immediate future of P),*
3. *$F^k(P) = F^{k-1}(F(P))$ (future of P after k steps of computation),* **if** $k > 1$.

The following remarks and lemmas give some properties of linked paths and their futures.

Remark 2. Let $\gamma_i \mapsto \gamma_{i+1}$ be a step. Let P be a linked path in γ_i. $\forall p \in V$, p hooks on to P in $\gamma_i \mapsto \gamma_{i+1}$ if and only if p executes Action F in $\gamma_i \mapsto \gamma_{i+1}$ and $p = FE(F(P))$ in γ_{i+1}. As Par_r is a constant equal to $\perp$, r cannot hook on to any linked path.

Remark 3. Let $\gamma_i \mapsto \gamma_{i+1}$ be a step such that there exists a linked path P in γ_i. A processor p unhooks from P in $\gamma_i \mapsto \gamma_{i+1}$ in the three following cases only:

1. P is an abnormal linked path, $IE(P) = p$ and p executes Action C,
2. $S_p = done$ and its parent in P executes Action B ($p \neq r$),
3. $p = r$, its child q satisfies $S_q = done$, and r sets S_r to *done* by executing Action B. In this case, q is also unhooked from P (Case 2.); moreover, since r never has any parent, $IE(P) = r$ and setting S_r to *done* involves that P disappears, i.e., $F(P) = Dead_P$.

The following lemma allows us to claim that, during a computation, the identities of processors which hook on to a linked path P and its future are included into the $Visited$ set of the final extremity of the future of P. By checking Actions B and F of Algorithms 1. and 2., this lemma is easy to verify:

Lemma 3. *Let P be a linked path. While $F^k(P) \neq Dead_P$ (with $k \in \mathbb{N}$), $Visited_{FE(F^k(P))}$ contains exactly $Visited_{FE(P)}$ union the identities of every processor which hooks on to P and its future until $F^k(P)$.*

By checking Action F of Algorithms 1. and 2., follows:

Lemma 4. *For all linked path P, $\forall\ p \in V$ such that $Id_p \in Visited_{FE(P)}$, p cannot hook on to P.*

By Lemmas 3 and 4, we deduce the next lemma.

Lemma 5. *For all linked path P, if $p \in V$ hooks on to P, then p cannot hook on to $F^k(P)$, $\forall k \in \mathbb{N}^+$.*

In the rest of the paper, we study the evolution of the paths. So, a lot of results concern P and $F^k(P)$ with $k \in \mathbb{N}$. From now on, when there is no ambiguity, we replace "P and $F^k(P)$, $\forall k \in \mathbb{N}$" by P only.

4.2 Proof Assuming a Weakly Fair Daemon

Now, we assume a weakly fair daemon. Under this assumption, the number of steps of any round is finite. So, as we have defined the future of a linked path in terms of steps, we can also evaluate the future of a linked path in terms of rounds. Let $e \in \mathcal{E}$. Let P be a linked path in γ_i ($\in e$). We note $F_R^K(P)$ the future of P, in e, after K rounds from γ_i.

We now show that the network contains no abnormal linked path in at most N rounds, i.e., every abnormal path P of the initial configuration satisfies $F_R^N(P)$ $= Dead_P$.

Theorem 2. *The system contains no abnormal linked path in at most N rounds.*

Sketch of Proof. It is easy to see that the number of abnormal linked paths cannot increase. Moreover, if Action C is enabled at p, then it remains enabled until p executes it. So, let P be an abnormal linked path. As the daemon is weakly fair, after each round, at least one processor unhooks from P (while P exists). By Lemmas 1, 4 and 5 and Remark 2, the number of processors which can hook on to P is at most $N - length(P)$. So, in the worst case, N rounds are necessary to unhook the processors of P and those which will hook on. Thus, $F_R^N(P) = Dead_P$. □

The following lemmas and theorems allow to prove that r eventually executes Action F.

Lemma 6. *For every normal linked path P, the future of P is $Dead_P$ after at most $2N - 2$ actions on it.*

Proof. Let $e \in \mathcal{E}$. Let $\gamma_i \in e$. Assume that there exists a normal linked path P in γ_i. First, we can remark that the future of P is either a normal linked path or $Dead_P$. Moreover, obviously, each action on P is either Action F or Action B. By Lemmas 4 and 5, only processors p such that $Id_p \notin Visited_{FE(P)}$ (in γ_i) can hook on to P at most one during the execution. By Lemma 2, in the worst case, the number of processors which hook on to P during the execution is $N - length(P)$. Then, after $N - 2$ processors unhooked from P (i.e., $length(P)$ $+$ $(N - length(P))$ $-$ 2 actions B on P), P satisfies $length(P) = 2$. In this case, only one action can be executed on P: the parent of $FE(P)$ (i.e., $IE(P)$) can execute Action B. Now, by Lemma 3, $Visited_{FE(P)} = \{Id_q :: q \in V\}$. So, by executing Action B, $IE(P)$ sets $S_{IE(P)}$ to $done$ $(Next_{IE(P)})$. Thus, as explained in Remark 3, P disappears. Hence, in the worst case, the future of P is $Dead_P$ after $N - length(P) + (N - 2) + 1$ actions which is maximal if initialy $length(P) = 1$, i.e., $2N - 2$ actions. □

If there exists no abnormal linked path, we can remark that, after at most one round, there always exists at least one continuously enabled action on the normal linked path. Thus, by Lemma 6 follows:

Lemma 7. *Let P be a normal linked path. If there exists no abnormal linked path, $F_R^{2N-1}(P) = Dead_P$.*

Theorem 2 and Lemma 7 prove the following theorem.

Theorem 3. *For all normal linked path P, $F_R^{3N-1}(P) = Dead_P$.*

Theorem 4. *From any initial configuration, r executes Action F after at most $3N$ rounds.*

Proof. By Theorems 2 and 3, from any initial configuration, the system needs at most $3N-1$ rounds to reach a configuration γ_i satisfying $\forall p \in V$, $S_p \in \{idle, done\}$. In γ_i, $\forall p \in V$ such that $S_p = done$, we have, $S_{Par_p} \neq p$. So, every p has Action C continuously enabled. As the daemon is weakly fair, after one round, $\forall p \in V$, $S_p = idle$. Thus, r is the only enabled processor and Action F is the only enabled action of r. Hence, from any initial configuration, the root executes Action F after at most $3N$ rounds. □

From the explanation provided in Section 3, it is easy to verify that when the system starts from a configuration where $\forall p \in V$, $S_p = idle$ (let us call it the *idle* configuration) it performs a traversal of the network according to Specification 1. Now, if the system starts from an arbitrary configuration, then it can contain some pre-clean processors and abnormal linked paths. We can remark that the pre-clean processors and the abnormal linked paths can only slow down the progression of the normal linked path. But the system keeps even so a normal behavior because the normal linked path progresses in the same way than if it starts from a *idle* configuration. So, the normal linked path eventually visits all the processors in the $first\ DFS$ order and, after, r eventually detects the termination of the wave when r sets S_r to *done* (because $\forall p \in V$, $Id_p \in Visited_r$). Hence:

Theorem 5. *From any configuration where r executes Action F, the execution satisfies Specification 1.*

From Remark 1, Theorems 4 and 5, follows:

Theorem 6. *Algorithm snap$\mathcal{DFS}$ is snap-stabilizing for Specification 1 with a weakly fair daemon.*

4.3 Proof Assuming an Unfair Daemon

From now on, we do not make any fairness assumption. The two next lemmas allow to prove that, in any execution of Algorithm *snap$\mathcal{DFS}$*, each round is finite.

Lemma 8. *The future of an abnormal linked path P is $Dead_P$ after at most $2N-1$ actions on it.*

Proof. The reasonning is similar to the proof of Lemma 6. □

Lemma 9. *Every round of Algorithm snap$\mathcal{DFS}$ has a finite number of steps.*

Proof. Let $e \in \mathcal{E}$. Assume that a round R of e has an infinite number of steps. Let γ_R be the first configuration of R.

First, assume that some abnormal linked paths of γ_R never disappear. So, the system eventually reaches a configuration $\gamma_i \in R$ in which there exists only abnormal linked paths which never disappear. Now, as every abnormal linked path disappears after a finite number of actions on it (see Lemma 8), there exists a configuration γ_j ($j \geq i$) from which no action will be executed on these abnormal linked paths. Then, every pre-clean processor is clean after one Action C and a normal linked path can only generate a finite number of pre-clean processors. Indeed, the pre-clean processors generated by the normal linked path has belong to it before and, until the normal linked path disappears, only a finite number of processors can hook on to it (see Lemma 5). Then, the pre-clean processors cannot prevent forever actions to be executed on a normal linked path. Now, by Lemma 6, every normal linked path disappears after a finite number of steps. So, the root processor executes Action F infinitively often to create normal linked paths. But, if r executes Action F, then, by Theorem 5, r creates a new normal linked path P and every processor ($\neq r$) eventually hooks on to P during the execution (in particular, the processors of abnormal linked paths). Now, a processor p can hook on to P if $S_p = idle$ (see Remark 2 and Predicate $Forward(p)$). Thus, P is eventually locked because the processors of the abnormal linked paths never hook on to it. So, r cannot execute Action F infinitively often, a contradiction. Thus, there exists a step $\gamma_{j'} \mapsto \gamma_{j'+1}$ with $j' \geq j$ in which at least one action is executed on an abnormal linked path, a contradiction. Hence, the abnormal linked paths eventually disappear.

So, there exists a configuration γ_k in which there exists no abnormal linked path. From this configuration, there always exists at most one linked path, the normal linked path. Assume that there exists no normal linked path in γ_k. Then, after a finite number of steps, r executes Action F and creates a normal linked path P (in the worst case, after $O(N)$ Actions C, every pre-clean processor becomes idle and r is the only enabled processor). As explained above, the pre-clean processors cannot prevent forever actions to be executed on P. By Lemma 6, the future of P is $Dead_P$ after a finite number of actions on it. Now, by Theorem 5, before disappearing, every processor hooks on to it by executing Action F. So, Round R is eventually done, a contradiction. Finally, if there exists a normal linked path P' in γ_k, by a similar reasonning, after a finite number of steps, the future of P' is $Dead_{P'}$ and we retrieve the previous case, a contradiction.

Hence, after a finite number of steps, every enabled processor of γ_R has executed one action. □

By Theorems 1 and 6, and Lemma 9, the following theorem holds.

Theorem 7. *Algorithm snap$\mathcal{DFS}$ is snap-stabilizing for Specification 1 even if the daemon is unfair.*

4.4 Complexity Analysis

Space Complexity. By checking Algorithms 1. and 2., follows:

Lemma 10. *The space requirement of Algorithm snap$\mathcal{DFS}$ is $O(N \times \log(N) + \log(\Delta))$ bits per processor.*

Time Complexity.

Lemma 11. *From any initial configuration, r executes Action F in $O(N^2)$ steps.*

Proof. In the initial configuration, the system can contain $O(N)$ pre-clean processors and $O(N)$ linked paths. Then, every linked path can generate $O(N)$ pre-clean processors. Indeed, the pre-clean processors generated by a linked path has belong to it before and, until a linked path disappears, every processor can hook on to it at most once (see Lemma 5). Finally, every pre-clean processor cleans it by executing Action C. And, every linked path disappears after $O(N)$ actions on it (see Lemmas 6 and 8). Hence, in the worst case, after $O(N^2)$ steps, r is the only enabled processor and executes Action F in the next step. □

The following lemma can be deduced from Lemma 11.

Lemma 12. *From any initial configuration, a complete $fDFS$ wave is executed in $O(N^2)$ steps.*

By Lemma 7, and Theorems 2 and 4, we can deduce the following result.

Lemma 13. *From any initial configuration, a complete $fDFS$ wave is executed in at most $5N - 1$ rounds.*

5 Conclusion

We presented a snap-stabilizing depth-first search wave protocol for arbitrary rooted networks. The protocol does not use any pre-computed spanning tree but requires identities on processors. The snap-stabilizing property guarantees that as soon as the root initiates the protocol, every processor of the network will be visited in DFS order. After the end of the visit, the root eventually detects the termination of the process. Furthermore, as our protocol is snap-stabilizing, by definition, it is also a self-stabilizing protocol which stabilizes in 0 round (resp. 0 step). Obviously, our protocol is optimal in stabilization time. We also showed that the proposed protocol works correctly assuming an unfair daemon, i.e., assuming the weakest scheduling assumption. Finally, note that our protocol executes a complete traversal of the network in $O(N)$ rounds and $O(N^2)$ steps, respectively. The memory requirement of our solution is $O(N \times \log(N) + \log(\Delta))$ bits per processor. In a future work, we would like to design a snap-stabilizing DFS wave protocol (for arbitrary rooted networks) with a memory requirement independent of N.

Acknowledgements. We would like to thank the anonymous referees for their suggestions and contructives comments on the earlier version of the paper. Their suggestions have greatly enhanced the readability of the paper.

References

1. Tel, G.: Introduction to distributed algorithms. Cambridge University Press (Second edition 2001)
2. Dijkstra, E.: Self stabilizing systems in spite of distributed control. Communications of the Association of the Computing Machinery **17** (1974) 643–644
3. Bui, A., Datta, A., Petit, F., Villain, V.: State-optimal snap-stabilizing PIF in tree networks. In: Proceedings of the Forth Workshop on Self-Stabilizing Systems, IEEE Computer Society Press (1999) 78–85
4. Awerbuch, B.: A new distributed depth-first-search algorithm. Information Processing Letters **20** (1985) 147–150
5. Cheung, T.: Graph traversal techniques and maximum flow problem in distributed computation. IEEE Transactions on Software Engineering **SE-9(4)** (1983) 504–512
6. Collin, Z., Dolev, S.: Self-stabilizing depth-first search. Information Processing Letters **49(6)** (1994) 297–301
7. Huang, S., Chen, N.: Self-stabilizing depth-first token circulation on networks. Distributed Computing **7** (1993) 61–66
8. Datta, A., Johnen, C., Petit, F., Villain, V.: Self-stabilizing depth-first token circulation in arbitrary rooted networks. Distributed Computing **13(4)** (2000) 207–218
9. Johnen, C., Beauquier, J.: Space-efficient distributed self-stabilizing depth-first token circulation. In: Proceedings of the Second Workshop on Self-Stabilizing Systems. (1995) 4.1–4.15
10. Petit, F., Villain, V.: Color optimal self-stabilizing depth-first token circulation. In: I-SPAN'97, Third International Symposium on Parallel Architectures, Algorithms and Networks Proceedings, IEEE Computer Society Press (1997) 317–323
11. Petit, F.: Fast self-stabilizing depth-first token circulation. In: Proceedings of the Fifth Workshop on Self-Stabilizing Systems, Lisbonne (Portugal), LNCS 2194 (October 2001) 200–215
12. Petit, F., Villain, V.: Time and space optimality of distributed depth-first token circulation algorithms. In: Proceedings of DIMACS Workshop on Distributed Data and Structures, Carleton University Press (1999) 91–106
13. Cournier, A., Datta, A., Petit, F., Villain, V.: Enabling snap-stabilization. In: 23th International Conference on Distributed Computing Systems (ICDCS 2003). (2003) 12–19
14. Dolev, S., Israeli, A., Moran, S.: Uniform dynamic self-stabilizing leader election. IEEE Transactions on Parallel and Distributed Systems **8** (1997) 424–440