© The Author 2006. Published by Oxford University Press on behalf of The British Computer Society. All rights reserved.
For Permissions, please email: journals.permissions@oxfordjournals.org
Advance Access published on January 4, 2006 doi:10.1093/comjnl/bxh154

Snap-Stabilizing Depth-First Search on Arbitrary Networks

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

LaRIA, CNRS FRE 2733, Universit de Picardie Jules Verne, Amiens (France)
**Corresponding author: devismes@laria.u-picardie.fr*

A *snap-stabilizing protocol*, starting from any arbitrary initial configuration, always behaves according to its specification. 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 (a preliminary version of this work was presented in OPODIS 2004, 8th International Conference on Principles of Distributed Systems, Grenoble (France)).

Keywords: Distributed systems, fault-tolerance, stabilization, depth-first search, wave algorithms.

Received 24 March 2005; revised 22 September 2005

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 which performs distributed computations called *waves*. These waves are initiated by at least one processor, called *initiator*, and require the participation of each processor of the network before a particular event called *decision*.

In an arbitrary rooted network, a Depth-First Search (*DFS*) wave is initiated by a particular processor called *root*. In this wave, all the processors are sequentially visited in DFS 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 programmation, routing or synchronization.

The concept of *self-stabilization* [2] is the most general technique to design a system tolerating arbitrary transient faults. A self-stabilizing system, regardless of the initial states of the processors and messages initially in the links, is guaranteed to converge to the intended 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 self-stabilizing protocol which stabilizes in 0 time unit. Obviously, a *snap-stabilizing* protocol is optimal in stabilization time.

Related works. There exist several (non-self-stabilizing) distributed algorithms solving this problem, e.g. [4, 5, 6]. In the area of self-stabilizing systems, silent algorithms (i.e. algorithms converging to a fixed point) computing a *DFS* spanning tree for arbitrary rooted networks are given in [7, 8, 9]. Several self-stabilizing (but not snap-stabilizing) wave algorithms based on the *depth-first token circulation* (*DFTC*) have been proposed for arbitrary rooted networks, e.g. [10, 11, 12, 13]. All these papers have a stabilization time in $O(N \times D)$ rounds where N is the number of processors and D is the diameter of the network. The algorithms proposed in [11, 12, 13] attempted to reduce the memory requirement from $O(\log(N)+\log(\Delta))$ [10] to $O(\log(\Delta))$ bits per processor where Δ is the degree of the network. However, the correctness of all above algorithms is proven assuming a (weakly) fair daemon. Roughly speaking, a daemon is considered an adversary which tries to prevent the protocol from behaving as expected, and fairness means that the daemon cannot prevent forever a processor from executing an enabled action.

The first snap-stabilizing *DFTC* has been proposed in [14] for tree networks. In arbitrary networks, a *universal transformer* providing a snap-stabilizing version of any (neither self- nor snap-) protocol is given in [15]. Obviously, combining this protocol with any *DFTC* algorithm (e.g. [4, 5, 6]), 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, independent of the token progress. Therefore, the number of steps per wave cannot be bounded.

Contributions. In this paper, we present the first snap-stabilizing DFS wave protocol for arbitrary rooted networks assuming an unfair daemon i.e. assuming the weakest scheduling assumption. Using this protocol, a *DFS* wave is bounded by $O(N^2)$ steps. In contrast, for the previous solutions, the step complexity of the *DFS* waves cannot be bounded. Our protocol does not use any pre-computed spanning tree. However, it requires identities on processors and has a memory requirement of $O(N \times \log(N) + \log(\Delta))$ bits per processor. The snap-stabilizing property guarantees that as soon as a *DFS* wave is initiated by the root, every processor of the network will be visited in *DFS* order and, after the end of the visit, the root will detect the termination of the process.

Outline of the paper. The rest of the paper is organized as follows: in Section 2, we describe the model in which our protocol is written. In the same section, we give a formal statement of the DFS Wave Problem solved in this paper. In Section 3, we present our DFS 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

Network. We consider a *network* 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. Our protocols are *semi-uniform*, i.e. each processor (of the network) 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 the 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.

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

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 $\mathcal{C}$, be 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 $\gamma (\gamma \in \mathcal{C})$ if there exists an enabled action 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 $\mathcal{P}$ is a *maximal* sequence of configurations $e = (\gamma_0, \gamma_1, \ldots, \gamma_i, \gamma_{i+1}, \ldots)$ such that, $\forall i \geq 0$, $\gamma_i \mapsto \gamma_{i+1}$ (called a *step*) if γ_{i+1} exists, else γ_i is a terminal configuration. *Maximality* means that the sequence is either finite (and no action of $\mathcal{P}$ 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 by $\mathcal{E}$.

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 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 execute an action except if it is the only enabled processor.

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 is not enabled in γ_{i+1}, but did not execute any protocol action in $\gamma_i \mapsto \gamma_{i+1}$. (The disabling action represents the following situation: at least one neighbor of p changes its state in $\gamma_i \mapsto \gamma_{i+1}$, and this change effectively made the guard of all actions of p false.)

In order to compute the time complexity, we use the definition of *round* [16]. 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.

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 2.1. (Snap-stabilization for Wave Protocols). *Let $\mathcal{T}$ be a task, and $\mathcal{SP}_\mathcal{T}$ a specification of $\mathcal{T}$. A wave protocol*

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:
  $F$ :: $Forward(p) \wedge \neg LockedF(p)$    $\rightarrow$  $Visited_p := \{Id_p\}$; $S_p := Next_p$;
  $B$ :: $Backward(p) \wedge \neg LockedB(p)$   $\rightarrow$  $Visited_p := ChildVisited_p$; $S_p := Next_p$;
  $C$ :: $Clean(p) \vee Error(p)$               $\rightarrow$  $S_p := idle$;
```

$\mathcal{P}$ is snap-stabilizing for the specification $\mathcal{SP_T}$ if and only if:

- *(i) at least one processor (called* initiator*) eventually executes a particular action of $\mathcal{P}$ (called* initialization action*);*
- *(ii) the result obtained with $\mathcal{P}$ from this* initialization action *always satisfies $\mathcal{SP_T}$.*

Specification of the DFS Wave Protocol. Before giving the specification of the DFS Wave Protocol, we propose some definitions.

DEFINITION 2.2. (Path). *The sequence of processors $p_1, \ldots, p_k (\forall i \in [1 \ldots k], p_i \in V)$ is a path of $G = (V, E)$ if $\forall i \in [1 \ldots k-1]$, $(p_i, p_{i+1}) \in E$. The path $p_1, \ldots, p_k$ 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 2.3. (First Path). *For each elementary path of G from the root, $P = (p_1 = r), \ldots, p_i, \ldots, p_k$, we associate a word $l_1, \ldots, l_i, l_{k-1}$ (noted $word(P)$) where, $\forall i \in [1 \ldots k-1]$, p_i is linked to p_{i+1} by the edge labeled 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 2.4. (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 $word(fp(p)) \prec_{lex} word(fp(q))$.*

DEFINITION 2.5. (Computation Wave). *A computation $e \in \mathcal{E}$ is called a* wave *if and only if the three following conditions hold:*

- *(i) e is finite;*
- *(ii) e contains at least one decision event;*
- *(iii) every processor in the network executes at least one action during e.*

SPECIFICATION 1. (*fDFS* Wave). A finite computation $e \in \mathcal{E}$ is called a *fDFS* wave (i.e. *first DFS* wave) if and only if the two following conditions hold:

- (i) *e* is a wave with a unique initiator: *r*;
- (ii) during *e*, all the processors are sequentially visited following the first DFS order;
- (iii) *r* decides when all the processors have been visited.[2]

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 both these conditions:

- (i) *r* eventually initiates a *fDFS* wave;
- (ii) 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 $snap\mathcal{DFS}$ (Algorithms 1 and 2). We first present the principle of the protocol. Next, we describe the data structures and the normal behavior of Algorithm $snap\mathcal{DFS}$. Finally, we give some details about the error correction.

[2]This implies that *r* detects when all the processors have been visited.

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:
  $Forward(p)$        ≡  $(S_p = idle) \wedge (\exists q \in Neig_p :: (S_q = p))$
  $Backward(p)$       ≡  $(\exists q \in Neig_p :: (S_p = q) \wedge (Par_q = p) \wedge (S_q = done))$
  $Clean(p)$          ≡  $(S_p = done) \wedge (S_{Par_p} \neq p)$
  $NoRealParent(p)$   ≡  $(S_p \notin \{idle, done\}) \wedge \neg(\exists q \in Neig_p :: (S_q = p) \wedge (Par_p = q))$
  $SetError(p)$       ≡  $(S_p \neq idle) \wedge [(Id_p \notin Visited_p) \vee (\exists q \in Neig_p :: (S_p = q) \wedge (Id_q \in Visited_p))$
                         $\vee (\exists q \in Neig_p :: (S_q = p) \wedge (Par_p = q) \wedge \neg(Visited_q \subsetneq Visited_p))]$
  $Error(p)$          ≡  $NoRealParent(p) \vee SetError(p)$
  $ChildError(p)$     ≡  $(\exists q \in Neig_p :: (S_p = q) \wedge (Par_q = p) \wedge (S_q \neq idle) \wedge \neg(Visited_p \subsetneq Visited_q))$
  $LockedF(p)$        ≡  $(|Pred_p| \neq 1) \vee (\exists q \in Neig_p :: (Id_q \notin PredVisited_p) \wedge (S_q \neq idle)) \vee (Id_p \in PredVisited_p)$
  $LockedB(p)$        ≡  $(|Pred_p| \neq 1) \vee (\exists q \in Neig_p :: (Id_q \notin ChildVisited_p) \wedge (S_q \neq idle)) \vee Error(p) \vee ChildError(p)$
Actions:
  $F$ :: $Forward(p) \wedge \neg LockedF(p)$    →  $Visited_p := PredVisited_p \cup \{Id_p\}$; $S_p := Next_p$; $Par_p := (q \in Pred_p)$;
  $B$ :: $Backward(p) \wedge \neg LockedB(p)$   →  $Visited_p := ChildVisited_p$; $S_p := Next_p$;
  $C$ :: $Clean(p) \vee Error(p)$               →  $S_p := idle$;
```

Principle. The principle of the normal behavior of Algorithm $snap\mathcal{DFS}$ consists in a *DFTC* split into two phases:

- the *visiting phase* where the token visits all the processors of the network in the first *DFS* order;
- the *cleaning phase* which cleans the traces of the *visiting phase* so that the root is eventually ready to initiate a new token circulation.

Starting now from any arbitrary configuration, several tokens, each one corresponding to a particular traversal, may co-exist in the network. The problem for the token from the root is to detect if a neighboring processor which seems to be visited has really received this token or not. To avoid both deadlocks and the non-visit of some processors of the network, we will use a list storing the IDs of all processors visited by the token. Thus, if the next processor to visit (according to the first *DFS* order) is in an abnormal behavior, the token holder waits until the *error correction* cleans the state of the 'abnormal' processor before it sends it the token.

Normal behavior. In its normal behavior, Algorithm $snap\mathcal{DFS}$ uses three variables for each processor p:

(i) S_p designates the *successor* of p in the visiting phase (from r), i.e. the next processor to receive the token. 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),

(ii) $Visited_p$ is the set of processors which have been visited by the token during the visiting phase,

(iii) Par_p designates the processor from which p receives the token for the first time during the visiting phase (as r creates the token, Par_r is the constant $\perp$).

In the following, we refer to Figure 1 in order to explain the normal behavior. In this figure, we show variable values only when they are considered in the protocol.

Consider now 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 such configurations, every processor $q \neq r$ such that $S_q = done$ is enabled to perform its cleaning phase (see Predicate $Clean(q)$). Processor q performs its cleaning phase by executing Action C, i.e. it assigns $idle$ to S_q (as Processor 3 in Step $i \mapsto ii$ of Figure 1). Moreover, in this configuration, the root (r) is enabled to initiate a visiting phase by Action F (n.b. the visiting and cleaning phase works in parallel). Processor r initiates a visiting phase (Step $i \mapsto ii$ in Figure 1) 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, then, r is the only enabled processor and initiates a visiting phase. From this point on, a token is created and held by r.

When a processor $q \neq r$ such that $S_q = idle$ is pointed out with S_p by a neighboring processor p, then q waits until all its neighbors q' such that $S_{q'} = done$ and $Id_{q'} \notin PredVisited_q$ (here, $Visited_p$) execute their cleaning phases ($ii \mapsto iii$ in Figure 1). Thereafter, q can execute Action F (as Processor 1 in

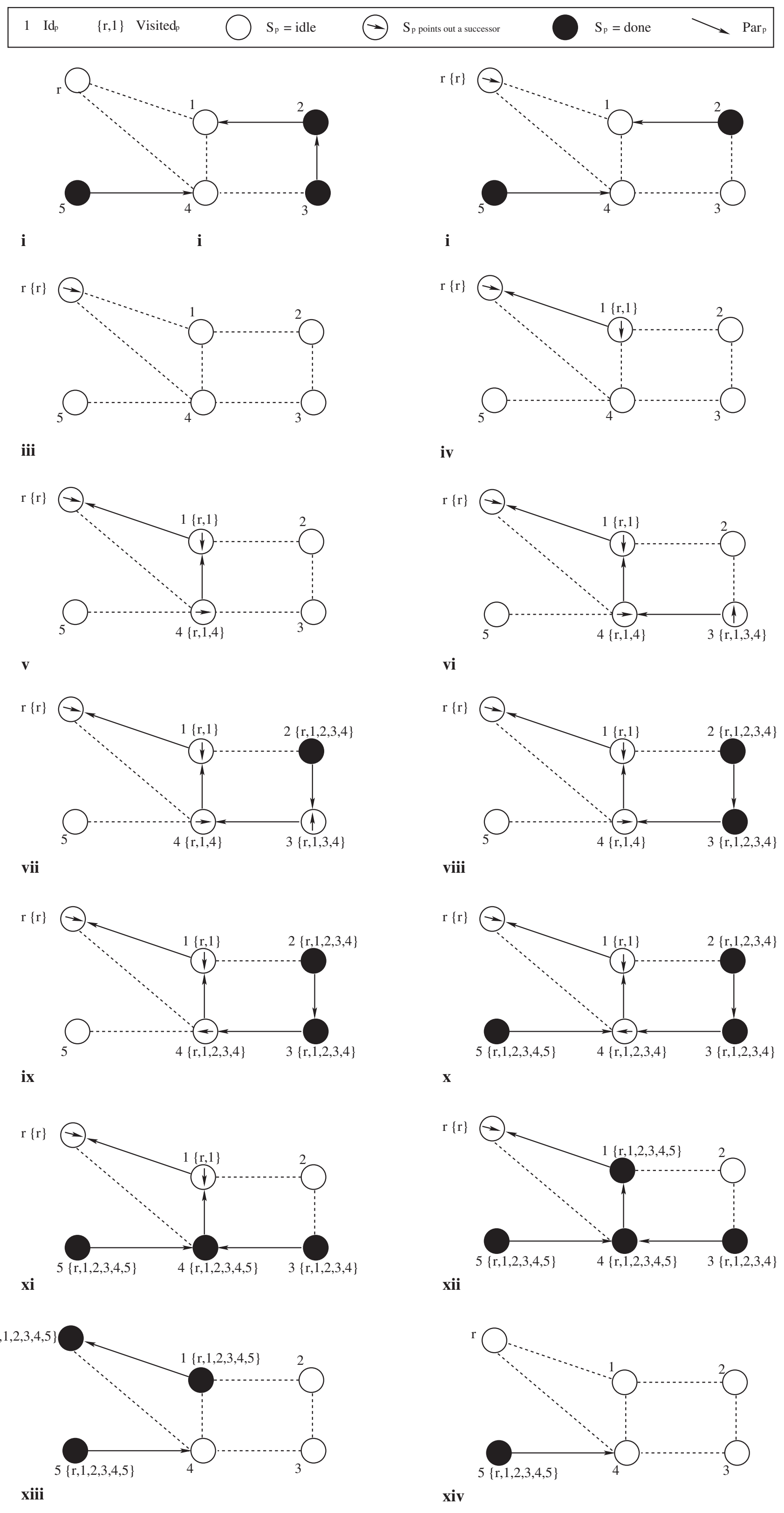

FIGURE 1. An example showing the normal behavior of Algorithm $snap\mathcal{DFS}$.

$iii \mapsto iv$, Figure 1) in order to receive the token from p. Then, q also designates p with Par_q and assigns $PredVisited_q \cup \{Id_q\}$ (here, $Visited_p \cup \{Id_q\}$) to $Visited_q$. Informally, the $Visited$ set of the last visited processor contains the identities of all processors visited by the token. Finally, q chooses a new successor, if any. For this earlier task, two cases are possible (see Macro $Next_q$):

(i) $\forall q' \in Neig_q, Id_{q'} \in Visited_q$, i.e. all neighbors of q have been visited by the token; the visiting phase from q is now terminated, so, S_q is set to $done$,

(ii) otherwise, q chooses the minimal processor by $\prec_q$ in $\{q' :: q' \in Neig_q \wedge Id_{q'} \notin Visited_q\}$ as a successor in the visit (i.e. the next processor to receive the token) .

In both cases, q is now the token holder.

When p is the predecessor of a processor q such that $S_q = done$, p has previously sent the token to q and knows that the visiting phase from q is now terminated. Thus, p must continue the visiting phase using another neighboring processor which is still not visited, if any: p executes Action B (e.g. Processor 3 in $vii \mapsto viii$ and Processor 4 in $viii \mapsto ix$, Figure 1). It assigns $ChildVisited_p$ to $Visited_p$. Hence, it knows exactly which processors have been visited by the token and it can designate another successor, if any, as in Action F (see Macro $Next_p$). By this action, p gets the token back and q is now enabled to execute its cleaning phase (Action C).

Eventually $S_r = done$ meaning that the visiting phase is terminated: the token has visited all the processors (Configuration $xiii$ in Figure 1) and so, r can execute its cleaning phase. Thus, the system eventually reaches a *normal initial configuration* again (Configuration xiv). Note that, since the cleaning phase does not erase parent pointers, the first *DFS* spanning tree of the network (w.r.t. Par variables) is available at the end of a single visiting phase.

Error correction. First, from the normal behavior, we can remark that, if $p \neq r$ is in the visiting phase (i.e. $S_p \neq idle$) and the visiting phase from p is still not terminated (i.e. $S_p \neq done$), 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:

(i) $(S_p \neq idle) \Rightarrow (Id_p \in Visited_p)$ because when p receives a token for the first time, it includes its identity in its *Visited* set (Action F);

(ii) $(S_p \in Neig_p) \Rightarrow (Id_{S_p} \notin Visited_p)$, i.e. p must not point out a previously visited processor;

(iii) $((p \neq r) \wedge (S_p \neq idle) \wedge (\exists q \in Neig_p :: (S_q = p) \wedge (Par_p = q))) \Rightarrow (Visited_q \subsetneq Visited_p)$ because while $p \neq r$ is in the visiting phase, $Visited_p$ must strictly include the $Visited$ set of its parent.

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 (i.e. the token) from p may 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 (the token holder) grows by the execution of Actions F and B and this processor can only extend the propagation using processors which are not in its *Visited* set. Hence, the visiting phase from an abnormal processor cannot run indefinitely and this abnormal visiting phase is 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 receive a token. 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:

(i) p has several predecessors;

(ii) p is an abnormal processor;

(iii) 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;

(iv) p $(S_p = idle)$ is designated as a successor by q but Id_p is in $Visited_q$, i.e. q is abnormal;

(v) 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

Proving a protocol with an unfair daemon is generally quite difficult. So, in the following, we will prove the protocol in two steps: in the first assuming a weakly fair daemon and in the second including the complement for the unfair daemon. This split is motivated by the following theorem:

THEOREM 4.1. *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 $\mathcal{SP_T}$ assuming a weakly fair daemon, e stabilizes to $\mathcal{SP_T}$. Hence, $\mathcal{P}$ is self-stabilizing for $\mathcal{SP_T}$ even if the daemon is unfair. □

Before presenting the proof of snap-stabilization, let us define some items and their characteristics.

4.1. Basic definitions and properties

Definition 4.1. (Idle Processor). *p is an idle processor if and only if $(S_p = idle)$.*

Definition 4.2. (Pre-clean Processor). *p is pre-clean if and only if $(Clean(p) \vee (S_p = done \wedge Error(p)))$.*

Definition 4.3. (Abnormal Processor). *p is abnormal if and only if $Error(p)$.*

Definition 4.4. (Linked Processors). *A processor p is linked to a processor q if and only if $(S_p = q) \wedge (Par_q = p) \wedge \neg SetError(q) \wedge (S_q \neq idle)$. p is called the parent of q. Respectively, q is called the child of p.*

Remark 2. Let $p, q \in V$ such that p is linked to q. Variable S_p (resp. Par_q) guarantees that q (resp. p) is the only child (resp. parent) of p (resp. q). As $Par_r = \perp$, by Definition 4.4, r never has any parent.

Definition 4.5. (Linked Path). *A linked path of G is a path $P = p_1, \ldots, p_k$ 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.*

For the next proof, we recall that a cycle is a path C satisfying $IE(C) = FE(C)$.

Lemma 4.1. *In any configuration, every linked path of G is elementary.*

Proof. Assume that there exists a non-elementary linked path P in an arbitrary configuration. Then, P contains a cycle $C = c_1, \ldots, c_k$. So, $c_1 = c_k$. From Definitions 4.4 and 4.5, $\forall\ i \in [1 \ldots\ k-1]$, c_i is linked to c_{i+1} i.e. $S_{c_i} = c_{i+1} \wedge Par_{c_{i+1}} = c_i \wedge S_{c_{i+1}} \neq idle$, and since $SetError(c_{i+1}) \equiv false$, $Visited_{c_i} \subsetneq Visited_{c_{i+1}}$. Now, because the inclusion relationship is transitive, $Visited_{c_1} \subsetneq Visited_{c_k}$. Hence, $Visited_{c_1} \subsetneq Visited_{c_1}$, a contradiction. □

From now on and until the end of the paper, we only consider maximal non-empty paths.

In the following lemma, we used the notion of distance $d(p, P)$ in a linked path. Let P be a linked path, let $p \in P$. We define $d(p, P)$ as follows:

- $d(p, P) = 0$, if $p = IE(P)$;
- $d(p, P) = 1 + d(q, P)$, where q is the parent of p in P, otherwise.

Lemma 4.2. *Every linked path P satisfies $Visited_{FE(P)} \supseteq \{Id_p :: p \in P \wedge p \neq IE(P)\}$.*

Proof. Let $\gamma_i \in \mathcal{C}$. Let P be a linked path in γ_i. To prove this lemma, we show that $\forall p \in P$, $Visited_p \supseteq \{Id_{p'} :: p' \in P \wedge d(p', P) \leq d(p, P) \wedge p' \neq IE(P)\}$ by induction on $d(p, P)$.

Let $f \in P$ such that $d(f, P) = 0$. By definition, $f = IE(P)$. As P is elementary (see Lemma 4.1) and maximal, there exists no processor $f' \in P$ such that $d(f', P) \leq d(IE(P), P) \wedge f' \neq IE(P)$. Thus, $\{Id_{f'} :: f' \in P \wedge d(f', P) \leq d(f, P) \wedge f' \neq IE(P)\} = \emptyset$ and the induction holds (trivially, $Visited_f \supseteq \emptyset$).

Assume now that $\forall p \in P$, such that $d(p, P) \leq d$ $(d \geq 0)$, p satisfies $Visited_p \supseteq \{Id_{p'} :: p' \in P \wedge d(p', P) \leq d(p, P) \wedge p' \neq IE(P)\}$.

Let $q \in P$ such that $d(q, P) = d + 1$. As $d(q, P) \geq 1$, by Definitions 4.4 and 4.5, $S_q \neq idle \wedge \neg SetError(q)$. Then, $Id_q \in Visited_q$. By Definition 4.5, $\exists p \in P$ such that p is linked to q. Moreover, $q \neq r$ (since $Par_r = \perp$, by Definition 4.4, no processor can be linked to r). Now, since $d(p, P) = d$, $Visited_p \supseteq \{Id_{p'} :: p' \in P \wedge d(p', P) \leq d \wedge p' \neq IE(P)\}$ (by assumption). Moreover, as $q \neq r \wedge S_q \neq idle \wedge \neg SetError(q)$, $Visited_p \subsetneq Visited_q$. Therefore, $Visited_q \supseteq (\{Id_{p'} :: p' \in P \wedge d(p', P) \leq d \wedge p' \neq IE(P)\} \cup \{Id_q\})$, i.e. $Visited_q \supseteq \{Id_{q'} :: q' \in P \wedge d(q', P) \leq d(q, P) \wedge q' \neq IE(P)\}$.

Thus, $\forall\ p \in P$ such that $d(p) \leq d + 1$, $Visited_p \supseteq \{Id_{p'} :: p' \in P \wedge d(p', P) \leq d(p, P) \wedge p' \neq IE(P)\}$. In particular, this property holds for $FE(P)$. □

Definition 4.6. (Normal and Abnormal Linked Paths). *A linked path P satisfying $Error(IE(P))$ is said to be abnormal. Respectively, we call normal linked path, every linked path which is not abnormal. Obviously, a normal linked path P satisfies $IE(P) =$ r.*

Lemma 4.3. *A normal linked path P satisfies $Visited_{FE(P)} \supseteq \{Id_p :: p \in P\}$.*

Proof. Let $\gamma_i \in \mathcal{C}$. Let P be a normal linked path in γ_i. By Definition 4.6, $IE(P) = r$. Moreover, by Definition 4.6, since $S_r \neq idle \wedge \neg Error(r)$, $Id_r \in Visited_r$. So, $Visited_{IE(P)} \supseteq \{Id_{IE(P)}\}$. Thus, we can prove this lemma by induction on $d(x, P)$ like in the proof of Lemma 4.2. □

We now introduce the notion of *future* of a linked path. We call the *future* of a linked path P the evolution of P during a computation. In particular, the *immediate future* of P is the

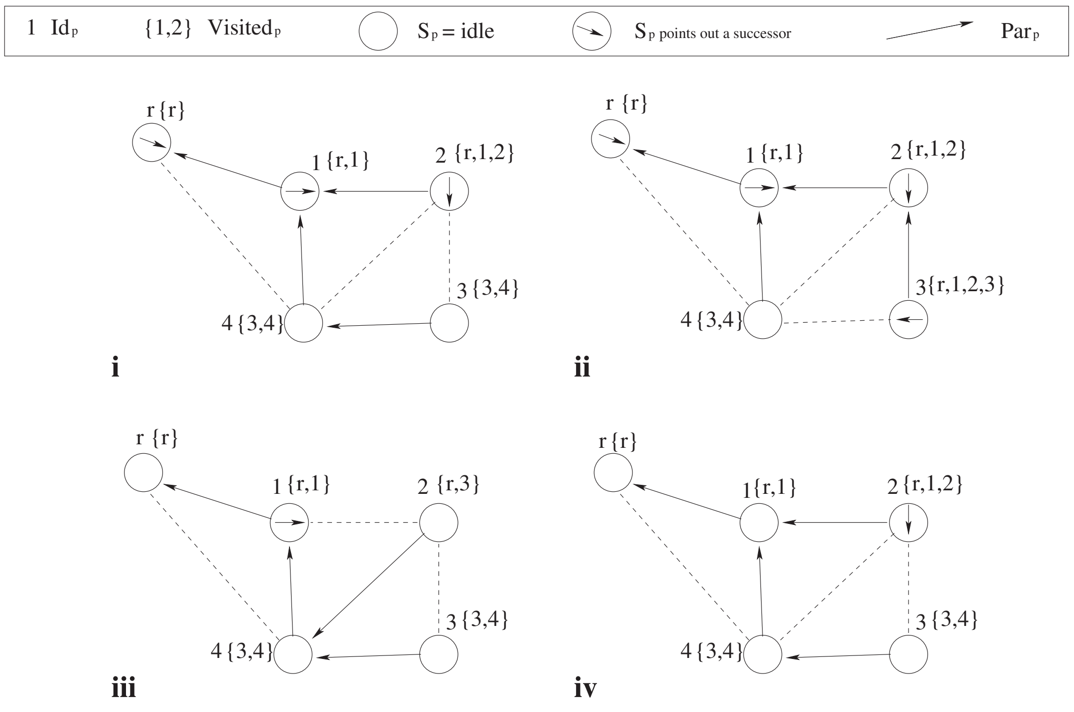

FIGURE 2. Instances of immediate futures.

transformation supported by P after one step. Note that, P may disappear after a step. Thus, by convention, we denote by $Dead_P$ the fact that P has disappeared after a step.

Definition 4.7. (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:*

(i) 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 $F(P) = P'$,
(ii) else, $F(P) = Dead_P$.

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

Figure 2 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 (i) (a) of Definition 4.7: 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 (i) (b) of Definition 4.7: in this execution, $F(P') = 2$. Note that if only Processor 1 executes Action C from iii, P' disappears, i.e. $F(P') = Dead_P$.

Definition 4.8. (Future of a Linked Path). *Let $e \in \mathcal{E}$, $\gamma_i \in e$, and P a linked path in γ_i. 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:*

(i) $F^0(P) = P$,
(ii) $F^1(P) = F(P)$ (immediate future of P),
(iii) $F^k(P) = F^{k-1}(F(P))$ (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 3. 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 4. 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:

(i) P is an abnormal linked path, $IE(P) = p$ and p executes Action C,
(ii) $S_p = done$ and its parent in P executes Action B ($p \neq r$),

(iii) $p = r$, 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 (ii)); moreover, since r never has any parent (see Remark 2), $IE(P) = r$ and setting S_r to $done$ involves that P disappears i.e. $F(P) = Dead_P$.

Remark 5. p ($\in V$) can unhook from the normal linked path only by executing Action B (by q with $Par_p = q$).

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 4.4. *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 $F^k(P)$.*

Lemma 4.5. *For all linked path P, $\forall\ p \in V$ such that $Id_p \in Visited_{FE(P)}$, p cannot hook on to P.*

Proof. Let $\gamma_i \mapsto \gamma_{i+1}$ be a step. Let P be a linked path in γ_i. By Remark 3, p ($\in V$) hooks on to P in $\gamma_i \mapsto \gamma_{i+1}$ by executing Action F. By $LockedF(p)$, $S_{FE(P)} = p$ and $Id_p \notin Visited_{FE(P)}$ in γ_i. □

By Lemmas 4.4 and 4.5, we deduce the next lemma.

Lemma 4.6. *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, many of the 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. Thus, we can reformulate Lemma 4.6 as follows: during any computation, a processor p cannot hook on to any linked path P more than once.

4.2. Proof assuming a weakly fair daemon

In this subsection, 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 $F_R^N(P) = Dead_P$.

The following remark is used in the proof of Lemma 4.7.

Remark 6. During the step $\gamma_i \mapsto \gamma_{i+1}$, Processor p can set S_p to q with $q \in Neig_p$ only if $S_q = idle$ in γ_i (see Predicates $LockedF(p)$ and $LockedB(p)$).

Lemma 4.7. *If Action C is enabled at p, it remains enabled until p executes it.*

Proof. Let $\gamma_i \mapsto \gamma_{i+1}$ be a step. Assume that a processor p has its action C enabled in γ_i and its action C is disabled in γ_{i+1}, but p does not execute it in $\gamma_i \mapsto \gamma_{i+1}$. Action C of p is enabled in γ_i if and only if $Clean(p) \vee ((p \neq r) \wedge NoRealParent(p)) \vee SetError(p)$. Moreover, as p had Action C enabled in γ_i, $S_p \neq idle$ in γ_i. So, we study the three following cases:

- $Clean(p)$ in γ_i. If $p = r$, then Action C is the only enabled action of r in γ_i. Moreover, $Clean(p)$ depends on variables of r only ($Clean(p) \equiv (S_p = done)$). So, if r does not execute Action C in $\gamma_i \mapsto \gamma_{i+1}$, then $S_r = done$ in γ_{i+1} and $Clean(p)$ remains $true$ in γ_{i+1}, a contradiction.

  If $p \neq r$, then $S_{Par_p} \neq p \wedge S_p \neq idle$ in γ_i. By Remark 6, if p does not move in $\gamma_i \mapsto \gamma_{i+1}$, then $S_{Par_p} \neq p \wedge S_p \neq idle$ in γ_{i+1}. Now, Action C is the only enabled action of p in γ_i. So, if p does not execute Action C in $\gamma_i \mapsto \gamma_{i+1}$, then Action C remains enabled in γ_{i+1}, a contradiction.
- $((p \neq r) \wedge NoRealParent(p))$ in γ_i. This case is similar to the previous case for $p \neq r$. So, we also obtain a contradiction.
- $SetError(p)$ in γ_i.

  Assume that $(Id_p \notin Visited_p) \vee (S_p \notin \{idle, done\} \wedge Id_{S_p} \in Visited_p)$. In this case, $SetError(p)$ is $true$ due to the variables of p only. Moreover, as $Error(p)$, Action C is the only enabled action of p in γ_i. So, p does not execute Action C in $\gamma_i \mapsto \gamma_{i+1}$. Then, $SetError(p)$ remains $true$ in γ_{i+1}, a contradiction.

  Hence, $SetError(p) \Rightarrow ((p \neq r) \wedge (\exists q \in Neig_p :: (S_q = p) \wedge (Par_p = q) \wedge \neg(Visited_q \subsetneq Visited_p)))$ in γ_i. We can remark that $SetError(p)$ depends on the states of p and q. Then, as $Error(p)$, Action C is the only enabled action of p in γ_i. So, while q does not execute any action, $SetError(p)$ remains $true$. Processor q can only execute Action C in $\gamma_i \mapsto \gamma_{i+1}$ (if $Error(q)$). Indeed, Action B of q is disabled because of $ChildError(q)$ and Action F of q is disabled because $S_q \neq idle$. Now, if q executes Action C in $\gamma_i \mapsto \gamma_{i+1}$, then $(Clean(p) \vee NoRealParent(p))$ is $true$ in γ_{i+1}. Hence, Action C remains enabled in γ_{i+1}, a contradiction. □

Theorem 4.2. *The system contains no abnormal linked path in at most N rounds.*

Proof. Let $e \in \mathcal{E}$. First, we can remark that the number of abnormal linked paths cannot increase in e. So, let P be an abnormal linked path of γ_0, the initial configuration of e. By Definition 4.6, in γ_0, $Error(IE(P))$ holds. By Lemma 4.7, Action C of $IE(P)$ is continuously enabled. As the daemon is weakly fair, $IE(P)$ eventually executes Action C in at most one round. Moreover, if $F_R^1(P) \neq Dead_P$, then $IE(F_R^1(P))$ will be continuously enabled and so on. Thus, after each round, at least one processor unhooks from P (while P does not disappear). By Lemmas 4.5 and 4.6, only processors p such that $Id_p \notin Visited_{FE(P)}$ (in γ_0) can hook on to P and they can do it at most once during the execution. Finally, by Remark 3 and Lemma 4.2, the number of processors which can hook on to P during the execution is at most $((N-1)-(length(P)-1))$, i.e. $(N - length(P))$. Then, in the worst case, N rounds are necessary (i.e. $length(P) + (N - length(P))$) to unhook the processors of P in γ_0 and those which will hook on. Thus, $F_R^N(P) = Dead_P$. Hence, after N rounds, the system contains no abnormal linked path. □

The following lemmas and theorems allow to prove that r eventually executes Action F (the initialization action).

LEMMA 4.8. *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, by Remarks 3 and 5, each action on P is either Action F or Action B. By Lemmas 4.5 and 4.6, only processors p such that $Id_p \notin Visited_{FE(P)}$ (in γ_i) can hook on to P and they can do it at most once during the execution. By Lemma 4.3, 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 4.4, $Visited_{FE(P)} = \{Id_q :: q \in V\}$. So, by executing Action B, $IE(P)$ sets $S_{IE(P)}$ to $done$ ($Next_{IE(P)}$). Thus, as explained in Remark 4, 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 initially $length(P) = 1$ i.e. $2N-2$ actions. □

LEMMA 4.9. *Let P be a normal linked path. If there exists no abnormal linked path, $F_R^{2N-1}(P) = Dead_P$.*

Proof. Let $e \in \mathcal{E}$. Let $\gamma_i \in e$ such that there exist only one linked path in γ_i: the normal linked path P. In every configuration reached from γ_i, the system contains no abnormal linked path. So, the system can only contain pre-clean processors, idle processors and a normal linked path (at most one). Now, all the pre-clean processors have Action C continuously enabled (see Lemma 4.7) and, as the daemon is weakly fair, they execute Action C in at most one round. Thus, after one round, if there exist some pre-clean processors, then they have been generated by the normal linked path. Hence, $\forall k \geq 1$, if $F_R^k(P) \neq Dead_P$, then the identity of every pre-clean processor q belongs to $Visited_{FE(F_R^k(P))}$ (see Remarks 4 and 5, and Lemma 4.4). So, $\forall k \geq 1$, if $F_R^k(P) \neq Dead_P$, two cases are possible at the beginning of the $k+1$th round from γ_i:

- $S_{FE(F_R^k(P))} = done$. Let p be the parent of $FE(F_R^k(P))$. First, there does not exist any abnormal linked path, so, $|Pred_p| = 1$. Then, from the above discussion and Lemma 4.3, every pre-clean processor and every processor of $F_R^k(P)$ belongs to $Visited_{FE(F_R^k(P))}$. Moreover, $ChildVisited_p = Visited_{FE(F_R^k(P))}$. Thus, $\forall q \in Neig_p$ such that $Id_q \notin ChildVisited_p$, $S_q = idle$. Hence, p satisfies $Backward(p) \wedge \neg LockedB(p)$ and Action B is continuously enabled because, except p, only pre-clean processors can execute an action: Action C.
- $S_{FE(F_R^k(P))} \neq done$. Let p such that $p = S_{FE(F_R^k(P))}$. As previously, $|Pred_p| = 1$ and $\forall q \in Neig_p$ such that $Id_q \notin PredVisited_p$, $S_q = idle$. Hence, p satisfies $Forward(p) \wedge \neg LockedF(p)$ and Action F of p is continuously enabled.

Then, at the beginning of each round, exactly one action on the normal linked path is continuously enabled until the normal linked path disappears. In the worst case, one action is executed on the normal linked path by a round. Thus, by Lemma 4.8, $F_R^{1+(2N-2)}(P) = Dead_P$, i.e. $F_R^{2N-1}(P) = Dead_P$. □

By Theorem 4.2 and Lemma 4.9, follows:

THEOREM 4.3. *For all normal linked paths P, $F_R^{3N-1}(P) = Dead_P$.*

THEOREM 4.4. *From any initial configuration,* r *executes Action F after at most $3N$ rounds.*

Proof. By Theorems 4.2 and 4.3, 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 (by Lemma 4.7). 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. □

The next theorem proves that, once initiated by r (Action F), the protocol behaves as expected.

THEOREM 4.5. *From any configuration where* r *executes Action F, Specification 1 holds.*

Proof. Assume that the system starts from a configuration where $\forall p \in V$, $S_p = idle$. Let us call it the $idle$ configuration. From such a configuration, r creates a normal linked path by Action F and this path is the only linked path in the network. So, each time a processor executes Actions F or B it makes progress to the visiting phase of the normal linked path. Now, each time a processor p executes Actions F or B it knows, thanks to $Visited_p$, which of its neighbors are visited or not (see explanations provided in Section 3). So, by Macro $Next_p$ of Actions F or B, either p sets S_p to $done$ because all its neighbors are visited (in this case, IDs of all its neighbors are in $\mathit{Visited}_p$) or p sets S_p to q such that q is the minimal non-visited neighbor of p by $\prec_p$ (i.e. the minimal neighbor of p by $\prec_p$ such that $Id_q \notin \mathit{Visited}_p$). Thus, starting from the $idle$ configuration, a traversal is performed in the network (by the normal linked path), this traversal follows the first DFS order (Definition 2.4), and the termination of the traversal is eventually detected by r when setting S_r to $done$. Hence, starting from the $idle$ configuration, the system runs according to Specification 1.

If the system starts from an arbitrary configuration, then it may contain some pre-clean processors and abnormal linked paths. But we now show that these pre-clean processors and abnormal linked paths can only slow down the progression of the normal linked path, P, (generated when r has executed Action F) and, despite these items, P even progresses in the network in the same way than if it starts from an $idle$ configuration. To that goal, assume that r executes Action F in $\gamma \mapsto \gamma'$. By Action F, r designates its minimal neighbor by $\prec_r$ as its successor (indeed, by $\neg LockedF(r)$, r executes Action F only if $\forall p \in Neig_r$, $S_p = idle$). Then, at each configuration reached from γ' (included γ'), two cases are possible according to $S_{FE(P)}$:

- $S_{FE(P)} = p$ i.e. $FE(P)$ designates p as successor. In this case, no processor of P is enabled while p does not hook on to it (Action F). Thus, P 'waits' until p hooks on to it. However, p cannot hook on to P while it satisfies one of the three following conditions:

  (i) p has several predecessors,
  (ii) p is not idle,
  (iii) p has one predecessor only but one of its non-visited neighbors is not idle. (By Lemma 4.4, with $\mathit{PredVisited}_p$, p knows the set of the visited processors by P.)

  Clearly, these conditions are satisfied due to the existence of pre-clean processors and/or abnormal linked paths. Now, we know that, on the one hand, the pre-clean processors eventually clean themselves (Action C is continuously enabled, see Lemma 4.7) and, on the other hand, the system does not contain any abnormal linked path in a finite time (Lemma 4.2). So, p eventually hooks on to P and p designates a successor among its non-visited neighbors (i.e. neighbors q such that $Id_q \notin \mathit{Visited}_p$), if any, w.r.t. $\prec_p$ (Macro Next_p).
- $S_{FE(P)} = done$. In the same way, when $FE(P)$ satisfies $S_{FE(P)} = done$, the visiting phase from $FE(P)$ is terminated and its parent q must continue the visiting phase by designating a new successor, if any, by Action B. Now, q is the only processor of P which can eventually execute an action (Action B) and it waits until all its non-visited neighbors become idle (with $\mathit{ChildVisited}_q$, q knows the set of the visited processors by P, see Lemma 4.4). Hence, like in the previous case, q eventually executes Action B: q designates a successor among its non-visited neighbors, if any, w.r.t. $\prec_q$ (Macro Next_q).

Thus, starting from any arbitrary configuration, despite the pre-clean processors and the abnormal linked paths that may exist, the normal linked path progresses in the same way than if it starts from the $idle$ configuration. Hence, after r executes Action F, the execution always satisfies Specification 1. □

From Remark 1, Theorems 4.4 and 4.5, follows:

THEOREM 4.6. *Algorithm $snap\mathcal{DFS}$ is snap-stabilizing for Specification 1 under a weakly fair daemon.*

4.3. Proof assuming an unfair daemon

We now show that our protocol works assuming an unfair daemon. In Section 4.2, we shown that it works assuming a weakly fair daemon (Theorem 4.6). So, according to Theorem 4.1, it remains to show that any execution of Algorithm $snap\mathcal{DFS}$ contains no infinite round.

LEMMA 4.10. *The future of an abnormal linked path P is $Dead_P$ after at most $2N - 1$ actions on it.*

Proof. Let $e \in \mathcal{E}$. Let $\gamma_i \in e$. Assume that there exists an abnormal linked path P in γ_i. First, we can remark that every future of P is either an abnormal linked path or $Dead_P$. As in the proof of Theorem 4.2, the number of processors which can hook on to P during the execution is at most $(N - length(P))$. Then, in the worst case, N processors must be unhooked from P, i.e. $length(P) + (N - length(P))$. So, N actions B or C must be executed on P (see Remark 4) to unhook the processors of P and those which will hook on. Thus, after these N actions, the future of P is $Dead_P$. Hence, in the worst case, the future of P is $Dead_P$ after $N - length(P)$ actions F (in the worst case, $length(P) = 1$) and N actions B or C, i.e. $2N - 1$ actions. □

LEMMA 4.11. *Every round of Algorithm $snap\mathcal{DFS}$ has a finite number of steps.*

Proof. Assume, by the contradiction, that there exists an execution $e \in \mathcal{E}$ containing an infinite round $R = (\gamma_0, \ldots, \gamma_i, \ldots)$.

Assume then that some abnormal linked paths existing in γ_0 never disapppear. So, the system eventually reaches a configuration $\gamma_i \in R$ in which there only exist abnormal linked paths which never disappear. Also, as every abnormal linked path disappears after a finite number of actions on it (Lemma 4.10), the system eventually reaches a configuration $\gamma_j \in R$ (with $j \geq i$) from which no actions are executed on these abnormal linked paths forever. From γ_j, the actions can only be executed on:

- *Idle processors.* An idle processor p can only execute Action F to hook on to the normal linked path or create a new normal linked path (when $p = r$).
- *Pre-clean processors.* A pre-clean processor can only execute Action C to clean it.

- *A normal linked path P.* P disappears after a finite number of actions on it (Lemma 4.8). Also, before disappearing, it generates a finite number of pre-clean processors only. Indeed, the pre-clean processors generated by P have belonged to it before and, until P disappears, only a finite number of processors can hook on to it (Lemma 4.6).

So, pre-clean processors cannot prevent forever actions to be executed on P (i.e. idle processors hooking on to P by Action F or processors of P executing Actions B). Now, by Lemma 4.8, P disappears after a finite number of actions on it. Hence, the only way to indefinitely extend the round R is that r executes Action F infinitively often. Now, between two actions F executed by r, the system satisfies Specification 1 (Theorem 4.5). In particular, this implies that the normal linked path P' created by r at the first execution of Action F has visited **all** the processors of the networks, especially, those in the abnormal linked paths. Now, a processor p of an abnormal linked path can hook on to P' only if $S_p = idle$ (see Remark 3 and Predicate *Forward*(p)), i.e. only if it has left its abnormal linked path by Action C. As we stated no action is executed on the abnormal linked path from γ_j, we obtain a contradiction. This contradiction implies that the previous assumption: 'some abnormal linked paths never disappear' is false.

So, there exists a configuration γ_k in which there exists no abnormal linked path. Of course, from this configuration, there always exists at most one linked path: the normal linked path. Similar to the above discussion, we can deduce that the only way to indefinitely extend the round R is that r executes Action F infinitively often. Now, between two Actions F executed at r, all processors are visited. So, each processor executes actions infinitively often. In particular, this implies that each enabled processor eventually executes an action (disabling action or protocol action). Hence, there exists no infinite round in e, a contradiction. □

By Theorems 4.6 and 4.1, and Lemma 4.11, the following theorem holds.

THEOREM 4.7. *Algorithm $snap\mathcal{DFS}$ is snap-stabilizing for Specification 1 assuming an unfair daemon.*

4.4. Complexity analysis

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

THEOREM 4.8. *The space requirement of Algorithm $snap\mathcal{DFS}$ is $O(N \times \log(N) + \log(\Delta))$ bits per processor.*

Time complexity. By Lemma 4.4, we already know that the delay to start a $fDFS$ wave is $O(3N)$ rounds. The next lemma gives us the complexity in terms of rounds to execute a complete $fDFS$ wave. It follows from Lemma 4.9, Theorems 4.2 and 4.4.

THEOREM 4.9. *From any initial configuration, a complete $fDFS$ wave is executed in at most $5N - 1$ rounds.*

Since Algorithm $snap\mathcal{DFS}$ is proven under the unfair daemon, we can now evaluate its step complexities.

THEOREM 4.10. *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 have belonged to it before and, until a linked path disappears, every processor can hook on to it at most once (see Lemma 4.6). Finally, every pre-clean processor cleans it by executing Action C. And, every linked path disappears after $O(N)$ actions on it (see Lemmas 4.8 and 4.10). Hence, in the worst case, after $O(N^2)$ steps, r is the only enabled processor and executes Action F in the next step. □

THEOREM 4.11. *From any initial configuration, a complete $fDFS$ wave is executed in $O(N^2)$ steps.*

Proof. The reasoning is similar to the proof of Theorem 4.10. □

Finally, we prove that the complexity result obtained in Lemma 4.10 is due to the first *fDFS* wave only. Indeed, as shown below, the time complexity of the other waves is $O(N)$ steps instead of $O(N^2)$.

LEMMA 4.12. *After the first $fDFS$ wave, the system contains no abnormal linked path.*

Proof. Assume that some abnormal linked paths does not disappear during the first $fDFS$ wave. Let P be one of this paths and P' be the normal linked path initiated by r at the beginning of the first $fDFS$ wave.

- Assume that P does not progress anymore, i.e. no processor hooks on to it. Then, as P' visits the processors in $first\ DFS$ order (By Theorem 4.7, Algorithm $snap\mathcal{DFS}$ satisfies Specification 1), P' eventually forces the processors of P to unhook from it in order to visit them. So, P eventually disappears (i.e. its future is eventually $Dead_P$), a contradiction.
- Assume that some processor hooks on to P during the first $fDFS$ wave.

Assume now that only processors which have been not visited by P' hook on to P. As the number of these processors decreases (by Theorem 4.7, Algorithm $snap\mathcal{DFS}$ satisfies Specification 1), P eventually does not progress anymore. So, as explained before, P' eventually forces the processors of P to unhook from it in order to visit them and P eventually disappears, a contradiction.

So, some processor visited by P' (during the first $fDFS$ wave) eventually hooks on to P. Let q be the first processor visited by P' which hooks on to P. Assume that q hooks on to P in $\gamma \mapsto \gamma'$. q satisfies $S_q = idle$ in γ (see Action F). So, q unhooked from P' in order to satisfy $S_q = idle$. By

Remark 4, q satisfies $S_p = done$ when it unhooked from P'. Moreover, q executes $S_q := done$ only if all its neighbors have been visited by P' (see Actions F and B, and Lemma 4.4). So, in γ, every neighbor of q has been already visited by P'. Now, in γ, $\forall p \in P$, p is not visited by P' (by assumption). In particular, $FE(P)$ is not visited by P' in γ but $FE(P) \in Neig_q$, a contradiction. $\square$

By Lemmas 4.8 and 4.12, the following result holds.

THEOREM 4.12. *After the first $fDFS$ wave, the other $fDFS$ waves are executed in $O(N)$ steps.*

5. CONCLUSION

We presented the first snap-stabilizing depth-first search wave protocol for arbitrary rooted networks assuming an unfair daemon, i.e. the weakest scheduling assumption. 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. Our protocol executes a complete traversal of the network in $O(N)$ rounds and $O(N^2)$ steps respectively. We also shown that after the first *DFS* wave, the other waves are executed in $O(N)$ steps only. 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.

REFERENCES

[1] Tel, G. (2001) *Introduction to Distributed Algorithms* (2nd edn). Cambridge University Press, Cambridge, UK.

[2] Dijkstra, E. (1974) Self stabilizing systems in spite of distributed control. *Commun. Assoc. Comput. Mach.*, **17**, 643–644.

[3] Bui, A., Datta, A., Petit, F. and Villain, V. (1999) State-optimal snap-stabilizing PIF in tree networks. In *Proc. Fourth Workshop on Self-Stabilizing Systems*, Austin, TX, June, pp. 78–85. IEEE Computer Society Press.

[4] Cheung, T. (1983) Graph traversal techniques and maximum flow problem in distributed computation. *IEEE Trans. Softw. Eng.*, **SE-9**(4), 504–512.

[5] Awerbuch, B. (1985) A new distributed depth-first-search algorithm. *Inform. Process. Lett.*, **20**, 147–150.

[6] Cidon, I. (1988) Yet another distributed depth-first-search algorithm. *Inform. Process. Lett.*, **26**, 301–305.

[7] Herman, T. (1991) Adaptivity through distributed convergence. PhD Thesis, Departement of Computer Science, University of Texas at Austin.

[8] Collin, Z. and Dolev, S. (1994) Self-stabilizing depth-first search. *Inform. Process. Lett.*, **49**(6), 297–301.

[9] Ducourthial, B. and Tixeuil, S. (2001) Self-stabilization with r-operators. *Distrib. Comput.*, **14**(3), 147–162.

[10] Huang, S. and Chen, N. (1993) Self-stabilizing depth-first token circulation on networks. *Distrib. Comput.*, **7**, 61–66.

[11] Johnen, C. and Beauquier, J. (1995) Space-efficient distributed self-stabilizing depth-first token circulation. In *Proc. Second Workshop on Self-Stabilizing Systems*, Las Vegas (UNLV), USA, May 28–29, pp. 4.1–4.15. Chicago Journal of Theoretical Computer Science.

[12] Johnen, C., Alari, C., Beauquier, J. and Datta, A. (1997) Self-stabilizing depth-first token passing on rooted networks. In *Proc. WDAG97 Distributed Algorithms 11th Int. Workshop*, Saarbrcken, Germany, September 24–26, *LNCS 1320*, pp. 260–274. Springer-Verlag.

[13] Datta, A., Johnen, C., Petit, F. and Villain, V. (2000) Self-stabilizing depth-first token circulation in arbitrary rooted networks. *Distrib. Comput.*, **13**(4), 207–218.

[14] Petit, F. and Villain, V. (1999) Time and space optimality of distributed depth-first token circulation algorithms. In *Proc. DIMACS Workshop on Distributed Data and Structures*, Princeton, USA, May 10–11, pp. 91–106. Carleton University Press.

[15] Cournier, A., Datta, A., Petit, F. and Villain, V. (2003) Enabling snap-stabilization. In *23th Int. Conf. Distributed Computing Systems (ICDCS 2003)*, Providence, RI, May 19–22, pp. 12–19. IEEE Computer Society Press.

[16] Dolev, S., Israeli, A. and Moran, S. (1997) Uniform dynamic self-stabilizing leader election. *IEEE Trans. Parall. Distrib. Sys.*, **8**, 424–440.