Snap-Stabilizing Detection of Cutsets

Alain Cournier, Stéphane Devismes, Vincent Villain
LaRIA, CNRS FRE 2733
Université de Picardie Jules Verne
Amiens, France

July 25, 2005

Abstract

A *snap-stabilizing protocol*, starting from any configuration, always behaves according to its specification. In this paper, we present a snap-stabilizing protocol which detects if a set of processors is a cutset of an arbitrary rooted network. This protocol is based on the depth-first search (DFS) traversal and its properties. An interesting property of our protocol is that, despite the initial configuration, as soon as the protocol is initiated (by the root), the result obtained from the computations will be right. So, after the first execution, the root is able to take a decision: "the input set is a cutset or not", and this decision is right.

keywords: Distributed systems, fault-tolerance, self-stabilization, snap-stabilization, cutset, separator.

1 Introduction

In this paper, we present the first snap-stabilizing protocol for detecting if a set of processors is a cutset of an arbitrary rooted network.

Consider a connected undirected graph $G = (V, E)$, where V is the set of N nodes and E the set of edges. $CS \subseteq V$ is a *cutset* (or a *separator*) of G if and only if the removal of all nodes of CS (and the incident edges) disconnects G. The detection of cutsets is an important issue in many applications such as evaluating the reliability of networks. Thus, from the fault tolerance point of view, detecting if a set of processors is a cutset of a network is essential.

The concept of *self-stabilization* [11] 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 initialy in the links, is guaranteed to converge to the intended behavior in a finite time. *Snap-stabilization* was introduced in [2]. 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. In the graph theory area, researchers are interested to scan all minimal cutsets of a graph. But, Provan and Ball proved that scanning all cutsets of a given graph in an NP-hard problem [20]. Thus, some heuristics have been designed for arbitrary graphs [19] and polynomial complete methods has developped for some particular class of graphs [1, 23]. Several works have been also proposed in distributed (non self-stabilizing) systems [13, 21]. In self-stabilizing systems, many algorithms has been written for finding cutnodes and/or bridges of a networks [4, 3, 17, 18, 10] (these problems are close to the cutsets detection

issue). But, to our best knowledge, nothing about cutsets has been proposed in self-stabilizing systems until now (in particular, neither in snap-stabilizing systems).

Contribution. In this paper, we present the first snap-stabilizing protocol for detecting if a set of processors is a cutset of an arbitrary rooted network. One of the most interesting properties of our protocol is that, despite the initial configuration, as soon as the protocol is initiated by the root, the result obtained from the computations will be right. So, after the first execution of the protocol, the root is able to take a decision: "the input set is a cutset or not", and this decision is right.

The presented protocol is the composition of a distributed cutset test algorithm with a previous snap-stabilizing DFS wave protocol [7]. The drawback of our solution is high cost memory requirement due to the snap-stabilizing DFS wave protocol. But, our cutset test algorithm may be composed with any self-stabilizing DFS wave protocol in order to improve the memory requirement. However, in this case, the resulting protocol will be self-stabilizing only.

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 Section 3, we present some useful properties about cutsets. Then, we present and prove our protocol in Sections 4 and 5. Finally, after presenting some complexity results (Section 6), we make concluding remarks (Section 7).

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 as 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 and is an input from the system.

Computational Model. In the computation model that we use, 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 in 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 $\mathcal{C}$ be the set of all possible configurations of the system. An action

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

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 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 *single computation step* or *move*) 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 their *enabled* actions. There exists several kinds of *daemon*. Here, we assume an *unfair distributed daemon*. The *unfairness* means that the daemon can forever prevent a processor to execute an action except if it is the only enabled processor. The *distributed* daemon implies that, 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.

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 order to compute the time complexity, we use the definition of *round* [12]. 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 first introduced in [2] as follows: a snap-stabilizing protocol guarantees that it always behaves according to its specification. In [5], authors discuss and formalize the definition to clarify the concept. In particular, they recall that snap-stabilization does not guarantee that all components of the system never work in a fuzzy manner. Snap-stabilization just ensures that if an execution of the protocol is initiated by some processor, then the protocol behaves as expected. The protocol we present is a *wave protocol* as defined by Tel in [22]. By definition, any execution of a wave protocol contains at least one initialization action. So, following [5], we propose a more simple definition of snap-stabilization holding for wave protocols.

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 $\mathcal{P}$ is snap-stabilizing for $\mathcal{SP}_{\mathcal{T}}$ if and only if:*

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

3 Basis of the Algorithm

3.1 Definitions

We now propose definitions of some terms of graph theory used in this paper.

Definition 3.1 (Path) *The sequence of processors $P = p_0, p_1, p_2, \ldots, p_k$ is a* path *of $G = (V, E)$ if $\forall i$, $1 \leq i \leq k$, $(p_{i-1}, p_i) \in E$. P is referred to as an* elementary path *if $\forall i,j$, $0 \leq i < j \leq k$, $p_i \neq p_j$. If*

p_0,p_1,p_2,. . .,p_{k-1} *is elementary and* $p_0 = p_k$*, then* P *is called a* cycle*. The processors* p_0 *and* p_k *are termed as the* extremities *of the path. The* length *of* P*, noted* $|P|$*, is the number of edges which compose* P*.*

Definition 3.2 (Subgraph) *$G_S = (V_S, E_S)$ is the subgraph of $G = (V, E)$ induced by V_S if and only if $V_S \subseteq V$ and $E_S = E \cap (V_S)^2$.*

Definition 3.3 (Connected Graph) *An undirected graph C is connected if and only if, for each pair of distinct nodes (p, q), there exists a path in C between p and q.*

Definition 3.4 (Connected Componante) *A connected componante of G is any connected subgraph of G maximal by inclusion.*

Definition 3.5 (Tree) *A connected undirected graph without any cycle is called a tree.*

Definition 3.6 (Spanning Tree) *A graph $T = (V_T, E_T)$ is a spanning tree of $G = (V, E)$ if and only if T is a tree, $V_T = V$, and $E_T \subseteq E$.*

Let $Tree(r) = (V, E_T)$ be a spanning tree rooted at r. We define the *height* of a node p in $Tree(r)$ (noted $h(p)$) as the length of the elementary path from r to p in $Tree(r)$. $H = \max_{p \in Tree(r)}\{h(p)\}$ represents the height of $Tree(r)$. For a node $p \neq r$, a node $q \in V$ is said to be the *parent* of p in $Tree(r)$ if and only if q is the neighbor of p (in $Tree(r)$) such that $h(p) = h(q) + 1$. Conversely, p is said to be the *child* of q in $Tree(r)$. A node p_0 is said to be an ancestor of another node p_k in $Tree(r)$ (with $k > 0$) if there exists a sequence of nodes p_0, ..., p_k such that $\forall\, p_i$, with $0 \leq i < k$, p_i is the parent of p_{i+1} in $Tree(r)$. Conversely p_k is said to be a *descendant* of p_0. We will note $Tree(p)$ the subtree of $Tree(r)$ rooted at p ($\in V$), i.e., the subgraph of $Tree(r)$ induced by p and its descendants in $Tree(r)$. Finally, we will call *tree edges* the edges of E_T and *non-tree edges* the edges of $E \setminus E_T$. We will call *non-tree neighbors* of p ($p \in Tree(r)$), nodes linked to p by a non-tree edge.

Definition 3.7 (DFS Spanning Tree[2]) *Tree(r) is a DFS spanning tree of G (rooted at r) if and only if Tree(r) is a spanning tree of G and $\forall$ (p, q) $\in V^2$ if p is a neighbor of q in G, then p $\in$ Tree(q) or q $\in$ Tree(p).*

From Definition 3.7, we can easily deduce the following remark:

Remark 3.1 *Let Tree(r) be a DFS spanning tree of G. Let (p, x, y) $\in V^3$. If $x \in Tree(p)$ and $y \in Neig_x$, then either $y \in Tree(p)$ or $p \in Tree(y)$.*

3.2 Approach

Let $CS \subseteq V$. Let $G' = (V', E')$ be the subgraph of G induced by $V' = V \setminus CS$. Let $Tree(r) = (V, E_T)$ be a DFS spanning tree of G rooted at r.

Our algorithm is based on the following definition.

Definition 3.8 *CS is a cutset of G if and only if there exists at least two connected componantes in G'.*

We now introduce the notion of $CCRoot$, i.e., a "root" of a connected componante of G' (see Figure 3.1).

Definition 3.9 (CCRoot) *We call $CCRoot$ of a connected componante C of G', a node $p \in C$ satisfying $h(p) \leq h(p'), \forall p' \in C$ (i.e., p is a node of C with the minimal height in $Tree(r)$).*

[2]This definition holds for undirected graphs only.

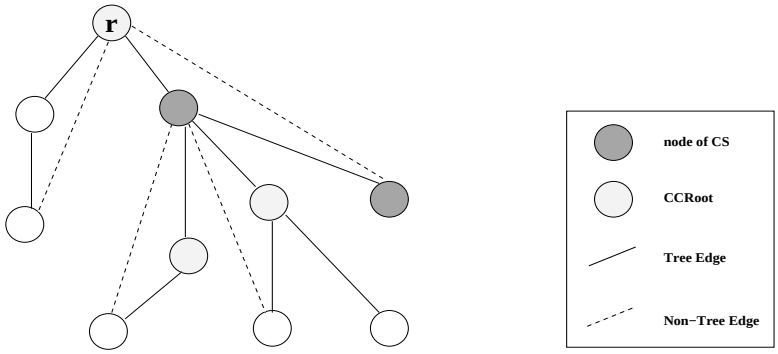

Figure 3.1: Example of $CCRoots$.

From Definition 3.9, we can easily deduce the following remark.

Remark 3.2 *If* r $\notin CS$*, then* r *is a* $CCRoot$ *of* C *where* C *is the connected componante of* r *in* G'.

The following lemmas allow to draw some useful properties for our protocol.

Lemma 3.1 *Let* C *be a connected componante of* G' *and* p *be a* $CCRoot$ *of* C. $Tree(p)$ *contains (at least) every node of* C.

Proof. By the contradiction. Assume that there exists some nodes of C which do not belong to $Tree(p)$. As C is connected, at least one of these nodes, y, is a neighbor of a processor x such that $x \in Tree(p)$. By Remark 3.1, either $y \in Tree(p)$ or $p \in Tree(y)$. So, by the contradiction, $p \in Tree(y)$. But, since p is a $CCRoot$ of C, by Definition 3.9, $h(y) \geq h(p)$. So, $h(y) = h(p)$ and $y = x$, a contradiction. □

Corollary 3.1 *For each connected componante of* G'*, there only exists one* $CCRoot$.

Remark 3.3 *As there exists exactly one* $CCRoot$ *per connected componante of* G' *(see Corollary 3.1), there is no ambiguity and we can simply write* $CCRoot$ *instead of* $CCRoot$ *of a connected componante of* G'.

From Definition 3.8 and Corollary 3.1, the following theorem is obvious.

Theorem 3.1 CS *is a cutset if and only if there exists at least two* $CCRoot$ *in* G'.

Thus, we need to (locally) detect if a node is a $CCRoot$. The following lemma provided a way to do that.

Lemma 3.2 *Let* C *be a connected componante of* G'. *A node* p *is the* $CCRoot$ *of* C *if and only if* p *satisfies the two following conditions:*

- $p \in C$.
- $\forall x \in Tree(p)$ *such that* $x \in C, \forall y \in Neig_x$: $y \notin CS \Rightarrow h(y) \geq h(p)$.

Proof.

- **If.** By the contradiction. Assume that, $p \in C$ and $\forall x \in Tree(p)$ such that $x \in C, \forall y \in Neig_x$: $y \notin CS \Rightarrow h(y) \geq h(p)$. But , p is not a $CCRoot$. So, by Definition 3.9, there exists some processors y' such that $y' \in C$ and $h(y') < h(p)$. In particular, $y' \notin Tree(p)$. But, as C is connected, among these processors, at least one y'' is a neighbor of a processor x' such that $x' \in Tree(p)$ and $x' \in C$. Now, by assumption, $h(y'') \geq h(p)$, a contradiction.

- **Only If.** Assume now that p is the $CCRoot$ of C. First, by Definition 3.9, $p \in C$ and $\forall q \in C$, $h(q) \geq h(p)$. Now, we can remark that $\forall x \in Tree(p)$ such that $x \in C, \forall y \in Neig_x$, if $y \notin CS$, then $y \in C$. Thus, $\forall x \in Tree(p)$ such that $x \in C, \forall y \in Neig_x$: $y \notin CS \Rightarrow h(y) \geq h(p)$.

□

4 Algorithm

In this section, we propose a snap-stabilizing protocol for detecting if a set of processors is a cutset of the network. Our protocol is the *conditional composition* of two other protocols: Algorithm $\mathcal{DFS}$ and Algorithm $\mathcal{CCRC}$ (the CCRoots Counting Algorithm). Algorithm $\mathcal{DFS}$[3] refers to the snap-stabilizing depth-first search (DFS) protocol of [7]. Algorithm $\mathcal{CCRC}$ uses the DFS properties in order to count the $CCRoot$ of the network as explained in the previous section. So, after recalling the definition of the *conditional composition*, we present Algorithm $\mathcal{DFS}$. We then introduce the data structures used by Algorithm $\mathcal{CCRC}$. Finally, we explain the behavior of the conditional composite algorithm $\mathcal{CCRCDFS}$, i.e., the conditional composition of Algorithm $\mathcal{CCRC}$ and Algorithm $\mathcal{DFS}$.

4.1 Conditional Composition

The *conditional composition* is a protocol composition technique which has been introduced by Datta et al in [8]. This general technique allows to simplify the design and proofs of Algorithm $\mathcal{CCRCDFS}$.

Definition 4.1 (Conditional Composition) *Let S_1 and S_2 be protocols such that variables written by S_2 are not referred by S_1. The conditional composition of S_1 and S_2, denoted by $S_2 \circ_{|\mathcal{G}} S_1$, is a protocol that satisfies the following conditions:*

1. *It contains all the variables and actions of S_1 and S_2.*
2. *$\mathcal{G}$ is a set of predicates and is a subset of the guards of S_1.*
3. *Every guard of S_2 has the form $g \wedge h$ or $\neg g \wedge h$ where g is a logical expression using the guards $\in \mathcal{G}$.*
4. *Since some actions of S_2 may also be enabled when an action of S_1 is enabled, the order of execution is the following: the action of S_2 followed by the action of S_1 (in the same step).*

4.2 Algorithm $\mathcal{DFS}$

We now roughly present Algorithm $\mathcal{DFS}$ (see [7, 6] for more details). In Algorithm $\mathcal{DFS}$, the root processor (r) eventually initiates a traversal of the network. During the traversal, all the processors are sequentially visited in DFS order. Algorithm $\mathcal{DFS}$ is snap-stabilizing. The snap-stabilizing property guarantees that, since r initiates the protocol, the traversal is performed as expected. In particular, the traversal cannot be corrupted by any abnormal behavior. The traversal performed by Algorithm $\mathcal{DFS}$ progresses in the network as a token circulation:

- The traversal begins when r creates a token by Action F.
- Each non-root processor p executes Action F when it receives the token for the first time.
- A processor p executes Action B each time the token is backtracked to it: If p has sent the token to q, then, since the traversal terminates at q (i.e., q holds the token and the token has visited all its neighbors), q backtracks the token to p.

[3] Algorithm $\mathcal{DFS}$ is provided in the appendix.

Obviously, the traversal performed by Algorithm $\mathcal{DFS}$ follows a DFS spanning tree of the network. Frow now on, we note $Tree(r) = (V, E_T)$ this tree. Also, we note $h(p)$ the height of the node p in $Tree(r)$ and H the height of $Tree(r)$.

4.3 Algorithm $\mathcal{CCRC}$

Algorithm $\mathcal{CCRC}$ is just an application of the properties shown in Section 3. Now, we describe the inputs, variables, and actions of Algorithm $\mathcal{CCRC}$.

Algorithm 4.1 Algorithm ($\mathcal{CCRC}$) CCRoots Counting for p = r

Input:
$Neig_p$: set of neighbors (locally ordered);
$S_p \in Neig_p \cup \{idle, done\}$: variable from Algorithm $\mathcal{DFS}$;
$Forward(p)$, $Backward(p)$, $LockedF(p)$, $LockedB(p)$: predicates from Algorithm $\mathcal{DFS}$;
$Next_p$: macro from Algorithm $\mathcal{DFS}$;
$InCS_p$: boolean;
Constant: $Level_p = 0$;

Variables: $IsCutset_p$: boolean; Cnt_p: integer;

Macros:
$InitCnt_p$ = **if** $(InCS)$ **then** $Cnt_p := 0$; **else** $Cnt_p := 1$;
$UpdIsCutset_p$ = **if** $(Next_p = done)$ **then** $IsCutset_p := (Cnt_p \geq 2)$;
Actions:
$Forward(p) \wedge \neg LockedF(p) \rightarrow InitCnt_p; UpdIsCutset_p;$
$Backward(p) \wedge \neg LockedB(p) \rightarrow Cnt_p := Cnt_{S_p}; UpdIsCutset_p;$

Inputs. Algorithm $\mathcal{CCRC}$ reads two inputs from Algorithm $\mathcal{DFS}$: S_p and $Next_p$[4]. The current successor (resp. predecessor) of a processor p in the traversal is maintained in S_p (resp. P_p). Note that $S_p \in Neig_p \cup \{idle, done\}$ meaning that p is ready to receive the token ($S_p = idle$), the traversal from p is done ($S_p = done$), or the traversal from p is in progress (and S_p designates its current successor in the traversal). Moreover, using the S variables, p can dynamically evaluate its parent P_p in $Tree(r)$ as follows: $P_p = q$ where $S_q = p$ (see Macro P_p). Finally, Macro $Next_p$ allows to compute a new value for S_p. In Algorithm $\mathcal{CCRC}$, we only use this macro to know when the traversal from p is done, i.e., when $Next_p = done$.

To simplify the design of the algorithm, we assume that every processor p knows if it belongs to the set to test (noted CS) thanks to the boolean $inCS_p$. In fact, we show $inCS_p$ as an input of the system but we could provided CS (using a set of Ids) in the input of r only and, after, propagated it to all the other processors using Algorithm $\mathcal{DFS}$.

Variables. In Algorithm $\mathcal{CCRC}$, every processor p maintains the following datas:

- $Level_p$, Cnt_p, and $IsCutset_p$ for $p = r$.
- $Level_p$, $Back_p$, and Cnt_p for $p \neq r$.

$Level_p$ refers to as the height of p in $Tree(r)$. In $Back_p$, we compute the value $UNNTC(p)$ (i.e., means the Uppermost Non-Tree Neighbor of $Tree(p)$ in C_p, where C_p is the connected componante of p in G'). **If** $p \in CS$, $UNNTC(p)$ = -1. **Otherwise**, p belongs to a connected componate of G', noted C_p, and $UNNTC(p)$ is equal to the minimal value among the height of each node of $Tree(p) \cap C_p$ and the height of their non-tree neighbors q such that $q \in C_p$. Formally, we defi ne $UNNTC(p)$ as follows:

Definition 4.2 ($UNNTC(p)$) *Let $p \in V \setminus \{r\}$. $UNNTC(p)$ corresponds to one the two following cases:*

[4] According to the conditional composition, $\mathcal{CCRC}$ can read Macro $Next$ of $\mathcal{DFS}$ because $Next$ involves $\mathcal{DFS}$ variables only.

Algorithm 4.2 Algorithm ($\mathcal{CCRC}$) CCRoots Counting for $p \neq r$

Input:
$Neig_p$: set of neighbors (locally ordered);
$S_p \in Neig_p \cup \{idle, done\}$: variable from Algorithm $\mathcal{DFS}$;
$Forward(p)$, $Backward(p)$, $LockedF(p)$, $LockedB(p)$: predicates from Algorithm $\mathcal{DFS}$;
$Next_p$: macro from Algorithm $\mathcal{DFS}$;
$InCS_p$: boolean;
Variables: Cnt_p, $Level_p$, $Back_p$: integers;

Predicate:

$IsCCRoot(p) \equiv (Back_p = Level_p)$

Macros:

P_p	=	$(q \in Neig_p :: S_q = p)$;
$NonCSAncLevel_p$	=	$\{x \in \mathbb{N} :: (\exists q \in Neig_p :: Level_q = x \wedge Level_q < Level_p \wedge \neg inCS_q)\}$;
$NonCSDescBack_p$	=	$\{x \in \mathbb{N} :: (\exists q \in Neig_p :: Back_q = x \wedge Level_q > Level_p \wedge \neg inCS_q)\}$;
$UpdBack_p$	=	**if** $(InCS_p)$ **then** $Back_p := -1$;
		else $Back_p := \min(\{Level_p\} \cup NonCSAncLevel_p \cup NonCSDescBack_p)$;
$UpdCnt_p$	=	**if** $(IsCCRoot(p))$ **then** $Cnt_p := Cnt_p + 1$;
$Update_p$	=	**if** $(Next_p = done)$ **then** $UpdBack_p$; $UpdCnt_p$;

Actions:

$Forward(p) \wedge \neg LockedF(p)$	$\rightarrow$	$Level_p := Level_{P_p} + 1$; $Cnt_p := Cnt_{P_p}$; $Update_p$;
$Backward(p) \wedge \neg LockedB(p)$	$\rightarrow$	$Cnt_p := Cnt_{S_p}$; $Update_p$;

- $UNNTC(p) =$ *-1*, **if** *$(p \in CS)$*
- $UNNTC(p) = min_{x \in Tree(p) \cap C_p}(\{h(x)\} \cup \{h(y) :: (y, x) \in E' \wedge h(y) < h(x)\})$ *where C_p is the connected componante of p in G' and E' the edge set of G',* **otherwise**.

The following theorem shows that if $Level_p$ and $Back_p$ are correctly evaluated (i.e., if $Level_p = h(p)$ and $Back_p = UNNTC(p)$), then we can locally detect if p is a $CCRoot$ or not.

Theorem 4.1 *$\forall p \in V \setminus \{r\}$, p is a $CCRoot$ if and only if $p \notin CS$ and $h(p) = UNNTC(p)$.*

Proof.

- **If.** Assume that $p \notin CS$ and $h(p) = UNNTC(p)$. By Definition 4.2, $h(p) = UNNTC(p)$ means that $\forall x \in Tree(p)$ such that $x \in C_p$, $\forall y \in Neig_x$ such that $y \in C_p$, $h(y) \geq h(p)$ and, by Lemma 3.2, p is a $CCRoot$.
- **Only If.** By the contradiction. Assume that p is a $CCRoot$ but $p \in CS$ or $h(p) \neq UNNTC(p)$. Since $p \notin CS$ (Definition 3.9), $h(p) \neq UNNTC(p)$. By Definition 4.2, $UNNTC(p) \leq h(p)$. So, by the contradiction, $UNNTC(p) < h(p)$. $UNNTC(p) < h(p)$ means that $\exists x \in Tree(p) \cap C_p$, $\exists (y, x) \in E'$ such that $h(y) < h(p)$. Now, $x \in C_p$ and Edge (x, y) exists in G'. So, $y \in C_p$ and, by Definition 3.9, $h(y) \geq h(p)$, a contradiction.

□

Thus, thanks to the $Level$ and $Back$ variables, we can locally detect the $CCRoots$. So, in addition, we use the Cnt variables to count the $CCRoots$ of the network. Finally, the boolean $IsCutset_r$ is used as a flag to mark if CS is a cutset or not.

Actions. Using the conditionnal composition, the actions of Algorithm $\mathcal{CCRC}$ are executed in the same step of Actions F and B of Algorithm $\mathcal{DFS}$ (see Definition 4.1). Action F is enabled at p when p satisfies $Forward(p) \wedge \neg LockedF(p)$. Respectively, Action B is enabled at p when p satisfies $Backward(p) \wedge \neg LockedB(p)$.

During a traversal, when Processor p receives the token for the first time (Action F), p can compute a value depending on it and its parents: a *prefix action*. In Algorithm $\mathcal{CCRC}$, the prefix action allows to

compute $Level_p$ for non-root processors and to initialise Cnt_p for the root (Remark 3.2 allows to determine if *r* is a $CCRoot$ or not). Then, when the traversal locally terminates at p (p executes Actions F or B while $Next_p = done$), p can calculate a result depending on it, its neighbors and/or its descendants: a *postfix action*. Indeed, in this case, $Tree(p)$ is entirely computed and the token has visited all the neighbors of p. In Algorithm $\mathcal{CCRC}$, the postfi x action allows to:

- Compute $Back_p$ for $p \neq r$. Indeed, when the traversal terminates at p, all its neighbors have computed their height and all its descendants have evaluated their $Back$ Variable.
- Update Cnt_p for $p \neq r$. As $Back_p$ and $Level_p$ are evaluated, by Theorem 4.1, p knows if it is a $CCRoot$ and, if necessary, it increments Cnt_p.
- Update $IsCutset_p$ for $p = r$. When the traversal terminates at *r*, the traversal is entirely done. So, *r* knows the number of $CCRoots$ of the network and, using Theorem 3.1, Macro $UpdIsCutset_p$ updates $IsCutset_p$ as well.

Finally, some actions of Algorithm $\mathcal{CCRC}$ have to be executed at each step of Algorithm $\mathcal{DFS}$ (when Actions F or B are executed). These actions allow to maintain in the Cnt variables the number of $CCRoots$ currently discovered.

4.4 Algorithm $\mathcal{CCRCDFS}$

Algorithm $\mathcal{CCRCDFS}$ is shown as Algorithm 4.3. Informally, Algorithm $\mathcal{CCRCDFS}$ works as follows. The root, *r*, begins the traversal by creating a token and initialises Cnt_r to 0 or 1 according to Remark 3.2. Then, each time a processor $p \neq r$ receives the token for the fi rst time, it initialises Cnt_p ($Cnt_p := Cnt_{S_p}$) and computes its height in $Level_p$. Each time the token is backtracked to a processor q, q updates Cnt_q. When the traversal terminates at q, q computes $Back_q$. Indeed, all its neighbors have computed their $Level$ variables and all its descendants have already computed their $Back$ variables. Thus, by Theorem 4.1, q can decide if it is a $CCRoot$ or not and updates Cnt_q as well. Finally, when the traversal is completely done (i.e., the token is backtracked to *r* and the token has visited all its neighbors), *r* can decide if CS (the set of nodes to test) is a cutset (according to Theorem 3.1) and updates $IsCutset_r$ as well.

Algorithm 4.3 Algorithm ($\mathcal{CCRCDFS}$) CCRoots Counting and Depth-First Search $\forall p \in V$

$\mathcal{CCRC} \circ |_{\{Forward, LockedF, Backward, LockedB\}} \mathcal{DFS}$

Thus, from any initial confi guration, after the end of a DFS traversal initiated by *r*, we obtain a confi guration similar to the one shown in Figure 4.2. In this exemple, $CS = \{1, 6, 8\}$. Informally, the system contains two $CCRoots$: *r*, 2. The root processor *r* is a $CCRoot$ because $r \notin CS$ (Remark 3.2). Processor 2 is a $CCRoot$ because $2 \neq r$, $2 \notin CS$, and $Level_2 = Back_2$. During the traversal, the Cnt variables count the number of $CCRoots$ of the network (here, equal to 2) and $IsCutset_r$ is set to true at the end of the traversal according to Theorem 3.1.

5 Proof of Correctness

In this section, we prove that Algorithm $\mathcal{CCRCDFS}$ (i.e., the conditional composition of Algorithm $\mathcal{DFS}$ and Algorithm $\mathcal{CCRC}$) is snap-stabilizing under an unfair daemon.

First, we can remark that Algorithm $\mathcal{CCRC}$ does not change the variables used by Algorithm $\mathcal{DFS}$. Moreover, no action of Algorithm $\mathcal{CCRC}$ can prevent any action of Algorithm $\mathcal{DFS}$ since, when an action of Algorithm $\mathcal{CCRC}$ is executed at p, it is done in the same step of an action of Algorithm $\mathcal{DFS}$ at p (because of the conditional composition). So, we can deduce the following remark.

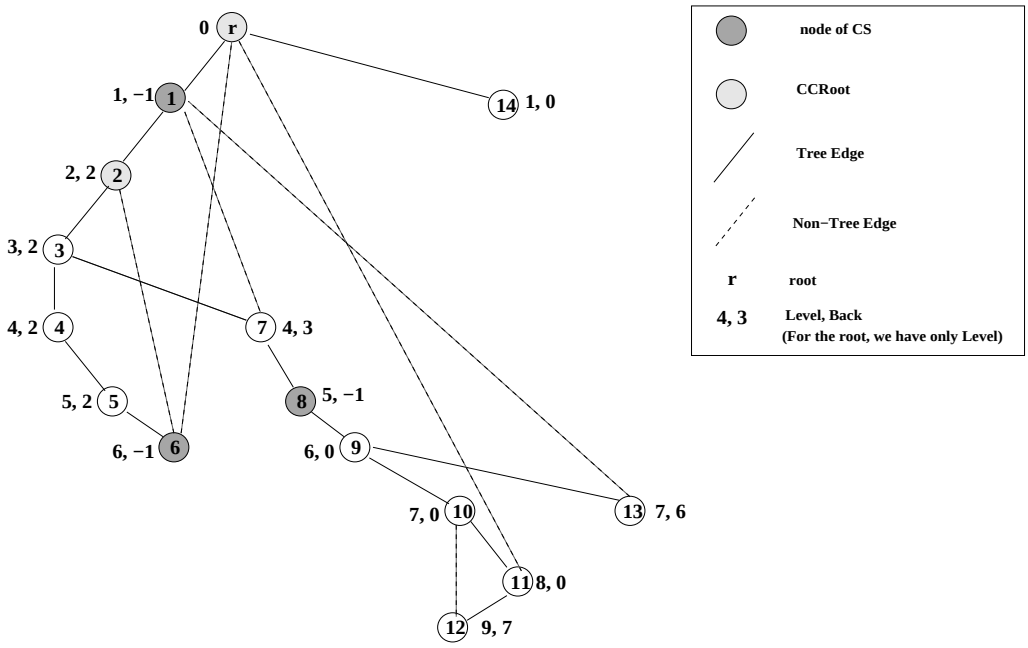

Figure 4.2: State of the network after the end of a DFS traversal initiated by *r*.

Remark 5.1 *Algorithm $\mathcal{CCRC}$ has no impact on the behavior of Algorithm $\mathcal{DFS}$.*

From [7], we know that Algorithm $\mathcal{DFS}$ is snap-stabilizing, i.e., *r* eventually initiates the protocol and since *r* initiates the protocol, Algorithm $\mathcal{DFS}$ satisfies its specification. More precisely, starting from any initial configuration, *r* eventually initiates a traversal of the network. During this traversal, all the processor are sequentially visited in DFS order. In particular, the snap-stabilizing property guarantees that the traversal performed by Algorithm $\mathcal{DFS}$ cannot be corrupted by any abnormal behavior. Moreover, we also know that Algorithm $\mathcal{CCRC}$ cannot prevent Algorithm $\mathcal{DFS}$ to work as expected (Remark 5.1). So, in the proofs, we will observe the system from the moment when *r* initiates the protocol and we focus on the traversal performed from *r* only (we do not take care of any abnormal behavior related to Algorithm $\mathcal{DFS}$).

The following lemma claims that, when a DFS traversal initiated by *r* is terminated, every processor p satisfies $Level_p = h(p)$.

Lemma 5.1 *$\forall p \in V$, after p receives the token (initiated by* r*) for the first time, p satisfies $Level_p = h(p)$ and $Level_p$ remains equal to $h(p)$ until the end of the traversal.*

Proof. We prove this lemma by induction on $h(p)$.

Let $p \in V$ such that $h(p) = 0$. In this case, $p = r$. Now, $Level_r$ is a constant equal to 0. So, trivially, the induction holds for $h(p) = 0$.

Assume now that, $\forall p \in V$ such that $h(p) \leq k$ with $0 \leq k < H$, after p receives the token for the first time, p satisfies $Level_p = h(p)$ and $Level_p$ remains equal to $h(p)$ until the end of the traversal.

Let $q \in V$ such that $h(q) = k + 1$. As $q \neq r$, q eventually receives the token for the first time from a processor q'. So, q' is the parent of q in $Tree(r)$ and $h(q') = k$. By induction assumption, $Level_{q'} = h(q')$. Now, q receives the token by executing Action F and, in the same step, by Algorithm $\mathcal{CCRC}$, q executes $Level_q := Level_{P_q} + 1$, i.e., $Level_{q'} + 1$. So, q sets $Level_q$ to $h(q)$. Moreover, until the end of the traversal,

$Level_q$ will be no more updated (see Algorithm $\mathcal{CCRC}$). Thus, the induction holds for $h(q) = k+1$ and the lemma holds. □

For the next lemma, we need to introduce the following notion.

Definition 5.1 (Distance) *Let $x \in V$. Let f be the leaf of $Tree(x)$ with the upper height. We define $d(x)$, the distance of x in $Tree(r)$, as follows: $d(x) = h(f) - h(x)$.*

With the following lemma, Lemma 5.1, and Theorem 4.1, $\forall p \in V \setminus \{r\}$, when the traversal (initiated by *r*) terminates at *p*, *p* is able to decide if it is a $CCRoot$ or not.

Lemma 5.2 *$\forall p \in V \setminus \{r\}$, when the traversal terminates at p, $Back_p = UNNTC(p)$.*

Proof. We prove this lemma by induction on $d(p)$.

Let $f \neq r$ be a leaf of $Tree(r)$ ($d(f) = 0$). When f receives the token (initiated by *r*) for the first time), f executes Action F (Algorithm DFS) and sets S_f to *done* ($Next_f = done$) meaning that the traversal from it is terminated. So, all neighbors of f have been visited by the token and, by Lemma 5.1, f and their neighbors have evaluated their height (in $Tree(r)$) in their $Level$ variable. In the same step of Action F, f executes Macro $Update_f$ (Algorithm $\mathcal{CCRC}$). Since $Next_f = done$, f executes Macro $UpdBack_p$:

- If $f \in CS$, then $InCS_f = true$ and f executes $Back_f := -1$. Thus, when the traversal terminates at f, $Back_f = UNNTC(f)$ (by Definition 4.2, page 7).
- If $f \notin CS$, then f executes $Back_f := \min(\{h(f)\} \cup NonCSAncLevel_f \cup NonCSDescBack_f$.
  $NonCSDescBack_f = \{h(x) :: x \in Neig_f \wedge x \notin CS \wedge h(x) > h(f)\}$. By Definition 3.7, $\forall x \in Neig_f$ such that $h(x) > h(f)$, $x \in Tree(f)$. Now, since f is leaf of $Tree(r)$, $Tree(f) = \emptyset$ and $NonCSDescBack_f = \emptyset$ also.
  $NonCSAncLevel_f = \{h(y) :: y \in Neig_f \wedge y \notin CS \wedge h(y) < h(f)\} = \{h(y) :: y \in Neig_f \wedge (y, f) \in E' \wedge h(y) < h(f)\}$ (remember that E' is the edge set of G').
  Let C_f be the connected componante of f in G'. As f is a leaf and $f \notin CS$, $Tree(f) \cap C_f = \{f\}$. So, after f receives the token, $Back_f = \min_{x \in Tree(f) \cap C_f}(\{h(x)\} \cup \{h(y) :: (y, x) \in E' \wedge h(y) < h(x)\})$ $= UNNTC(f)$ (by Definition 4.2, page 7).

Hence, the induction holds for $d(p) = 0$.

Assume now that, $\forall p \in V \setminus \{r\}$ such that $d(p) \leq k$ with $0 \leq k < H - 1$, when the traversal terminates at p, $Back_p = UNNTC(p)$.

Let $q \in V \setminus \{r\}$ such that $d(q) = k + 1$. The traversal from q terminates when q sets S_q to *done* by executing Action B (in Algorithm DFS). So, all the neighbors of q have been visited and by Lemma 5.1, q and their neighbors have evaluated their height in their $Level$ variable. In the same step of Action B, q executes Macro $Update_q$ (see Algorithm $\mathcal{CCRC}$). Since $Next_q = done$, q executes Macro $UpdBack_p$:

- If $q \in CS$, then $InCS_q = true$ and q executes $Back_q := -1$. Thus, when the traversal terminates at q, $Back_q = UNNTC(q)$ (by Definition 4.2).
- If $q \notin CS$, then q executes $Back_q := \min(\{h(q)\} \cup NonCSAncLevel_q \cup NonCSDescBack_q)$.
  $NonCSAncLevel_q = \{h(y) :: y \in Neig_q \wedge y \notin CS \wedge h(y) < h(q)\} = \{h(y) :: y \in Neig_q \wedge (y, q) \in E' \wedge h(y) < h(q)\}$.
  Let C_q be the connected componante of q in G'. $\forall p \in Neig_p$ such that $h(p) > h(q)$ and $p \notin CS$, $p \in C_q$ and, by Definition 3.7, $p \in Tree(q)$. So, $\forall p \in Neig_q$ such that $h(p) > h(q)$ and $p \notin CS$, $p \in Tree(q) \cap C_q$ and $d(q) \leq k$ (because $d(p) < d(q)$). Hence, from Algorithm 4.2 and by induction assumption, we can deduce that $NonCSDescBack_q = \bigcup_{(p \in Neig_q :: p \in C_q \cap Tree(q)} \{UNNTC(p)\}$. Hence, by Definition 4.2, $NonCSDescBack_q = \bigcup_{(p \in Neig_q :: p \in C_q \cap Tree(q))} \{\min_{x \in Tree(p) \cap C_q} (\{h(x)\} \cup \{ h(y) :: (y, x) \in E' \wedge h(y) < h(x)\})\}$.

So, $Back_q = \min(\{h(q)\} \cup \{h(y) :: y \in Neig_q \wedge y \in C_q \wedge h(y) < h(q)\} \cup (\bigcup_{(p \in Neig_q :: p \in C_q \cap Tree(q)}$ $\{ \min_{x \in Tree(p) \cap C_q}(\{h(x)\} \cup \{ h(y) :: (y,x) \in E' \wedge h(y) < h(x)\})\})) = \min_{x \in Tree(q) \cap C_q}(\{h(x)\} \cup$ $\{h(y) :: (y,x) \in E' \wedge h(y) < h(x)\}) = UNNTC(q)$ (by Definition 4.2).

Thus, the induction holds $\forall q' \in V \setminus \{r\}$ such that $d(q) \leq k+1$ and the lemma holds. □

The next lemma establishes the correctness of Algorithm $\mathcal{CCRCDFS}$.

Lemma 5.3 *At the end of a traversal initiated by r, $IsCutset_r = true$ if and only if CS is a cutset.*

Proof. From [7], we know that, in Algorithm $\mathcal{DFS}$, *r* eventually initiates a DFS traversal during what all the processors are sequentially visited in DFS order. By Remark 5.1, we know that actions of Algorithm $\mathcal{CCRC}$ cannot prevent the traversal to progress correctly. So, in Algorithm $\mathcal{CCRCDFS}$, *r* eventually creates a token. In the same step, *r* executes Macro $InitCnt_r$ in Algorithm $\mathcal{CCRC}$. By $InitCnt_r$, *r* decide, according to Remark 3.2, if it is a $CCRoot$. So *r* initialises Cnt_r as well (see Algorithm 4.1).

Then, the token traverses the network in DFS order. By Lemmas 5.1 and 5.2, when the traversal terminates at p such that $p \in V \setminus \{r\}$, p satisfies both $Level_p = h(p)$ and $Back_p = UNNC(p)$. So, according to Theorem 4.1, Predicate $IsCCRoot(p)$ determines if p is a $CCRoot$ and p updates Cnt_p as well by Macro $UpdBack_p$.

Moreover, every time the traversal progress, the token holder p (i.e., the processor which holds the token) maintains in Cnt_p the number of detected $CCRoots$ (see Algorithms 4.1 and 4.2).

Hence, when *r* detects the termination of the traversal, Cnt_r contains the number of $CCRoots$ of the network and, by Theorem 3.1, *r* decides if CS is a cutset and updates $IsCutset_r$ as well (see $UpdIsCutset_r$ in Algorithm 4.1). □

By Lemma 5.3, since it is initiated by *r*, Algorithm $\mathcal{CCRCDFS}$ works as expected. Then, in [7], Algorithm $\mathcal{DFS}$ is proven assuming an unfair daemon. Now, by Definition 4.1, Algorithm $\mathcal{CCRCDFS}$ works with the same number of steps than Algorithm $\mathcal{DFS}$. So, Algorithm $\mathcal{CCRCDFS}$ also snap-stabilizing and runs under the unfair daemon. Hence, the following theorem holds.

Theorem 5.1 *Algorithm $\mathcal{CCRCDFS}$ is snap-stabilizing and detects if CS is a cutset assuming an unfair daemon.*

6 Complexity Analysis

Time Complexity. Using the conditional composition, the actions of Algorithm $\mathcal{CCRC}$ are executed only when actions of Algorithm $\mathcal{DFS}$ are executed. Moreover, actions of Algorithm $\mathcal{CCRC}$ and Algorithm $\mathcal{DFS}$ are executed in the same step. Thus, the complexity results of Algorithm $\mathcal{CCRCDFS}$ and Algorithm $\mathcal{DFS}$ are the same. Hence, from [7], the following results are obvious. (For more details see [7, 6].)

Theorem 6.1 *From any initial configuration, a complete $\mathcal{CCRCDFS}$ computation is executed in at most $6N-1$ rounds.*

Theorem 6.2 *From any initial configuration, a complete $\mathcal{CCRCDFS}$ computation is executed in $O(N^2)$ moves.*

Space Complexity. In Algorithms 4.1 and 4.2, we do not assume any bound on Variables Cnt, $Level$, and $Back$. But, we may assume that the maximal value of each of these variables is any upper bound of N. Thus, we can claim that each variable Cnt, $Level$, or $Back$ can be stored in $O(\log N)$ bits and, by taking account of the other variables, follows:

Theorem 6.3 *The space requirement of Algorithm $\mathcal{CCRC}$ is $O(\log(N))$ bits per processor.*

From [7], we know that the space complexity of Algorithm $\mathcal{DFS}$ is $O(N \times \log(N) + \log(\Delta))$ bits per processor (where Δ is an upper bound on the degree of the processors). So, with Theorem 6.3, we can deduce:

Theorem 6.4 *The space requirement of Algorithm $\mathcal{CCRCDFS}$ is $O(N \times \log(N) + \log(\Delta))$ bits per processor.*

7 Conclusion

In this paper, we have presented the first snap-stabilizing protocol for detecting if a set of processors is a cutset of an arbitrary rooted network called Algorithm $\mathcal{CCRCDFS}$. This protocol, which is a conditionnal composition of Algorithms $\mathcal{CCRC}$ and $\mathcal{DFS}$, works assuming an unfair daemon, i.e., the weakest scheduling assumption. The snap-stabilizing property guarantees that despite the initial configuration, as soon as our protocol is initiated by the root, the result obtained from the computations will be right. Moreover, as our protocol is snap-stabilizing, by definition, it is also a self-stabilizing protocol which stabilizes in 0 round. Obviously, our protocol is optimal in stabilization time. In addition, note that a complete computation of Algorithm $\mathcal{CCRCDFS}$ is executed in $O(N)$ rounds and $O(N^2)$ moves. Finally, the space requirement of our solution is $O(N \times \log(N) + \log(\Delta))$ bits per processor. Algorithm $\mathcal{CCRC}$ can be combined with any self-stabilizing DFS wave protocol (e.g. [14, 16, 15, 9]) in order to improve the memory requirement. Of course, in this case, the resulting protocol will be self-stabilizing only.

References

[1] S H Ahmad. Simple enumeration of minimal cutsets of acyclic directed graph. *IEEE Transactions on Reliability*, 37:484–487, 1988.

[2] A Bui, AK Datta, F Petit, and V Villain. State-optimal snap-stabilizing PIF in tree networks. In *Proceedings of the Fourth Workshop on Self-Stabilizing Systems*, pages 78–85, Austin, Texas, USA, June 1999. IEEE Computer Society Press.

[3] P Chaudhuri. A note on self-stabilizing articulation point detection. *Journal of Systems Architecture*, 45(14):1249–1252, 1999.

[4] P Chaudhuri. An $o(n^2)$ self-stabilizing algorithm for computing bridge-connected components. *Computing*, 62:55–67, 1999.

[5] A Cournier, AK Datta, F Petit, and V Villain. Enabling snap-stabilization. In *23th International Conference on Distributed Computing Systems (ICDCS 2003)*, pages 12–19, Providence, Rhode Island USA, May 19-22 2003. IEEE Computer Society Press.

[6] A Cournier, S Devismes, F Petit, and V Villain. Snap-stabilizing depth-first search on arbitrary networks. Technical Report 2004-09, LaRIA, CNRS FRE 2733, 2004.

[7] A Cournier, S Devismes, F Petit, and V Villain. Snap-stabilizing depth-first search on arbitrary networks. In *OPODIS'04, International Conference On Principles Of Distributed Systems Proceedings*, pages 267–282, 2005.

[8] Ajoy Kumar Datta, Shivashankar Gurumurthy, Franck Petit, and Vincent Villain. Self-stabilizing network orientation algorithms in arbitrary rooted networks. In *International Conference on Distributed Computing Systems*, pages 576–583, 2000.

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

[10] S Devismes. A silent self-stabilizing algorithm for finding cut-nodes and bridges. *Parallel Processing Letters*, 15:183–198, 2005.

[11] EW Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17:643–644, 1974.

[12] S Dolev, A Israeli, and S Moran. Uniform dynamic self-stabilizing leader election. *IEEE Transactions on Parallel and Distributed Systems*, 8(4):424–440, 1997.

[13] N S Fard and T H Lee. Cutset enumeration of network systems with link and node failure. *Reliability Engineering and System Safety*, 65:141–146, 1999.

[14] ST Huang and NS Chen. Self-stabilizing depth-first token circulation on networks. *Distributed Computing*, 7:61–66, 1993.

[15] C Johnen, C Alari, J Beauquier, and AK Datta. Self-stabilizing depth-first token passing on rooted networks. In *WDAG97 Distributed Algorithms 11th International Workshop Proceedings, Springer-Verlag LNCS:1320*, pages 260–274, Saarbrücken, Germany, September 24-26 1997. Springer-Verlag.

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

[17] M Hakan Karaata. A self-stabilizing algorithm for finding articulation points. *International Journal of Foundations of Computer Science*, 10(1):33–46, 1999.

[18] M Hakan Karaata and P Chaudhuri. A self-stabilizing algorithm for bridge finding. *Distributed Computing*, 12(1):47–53, 1999.

[19] D R Karger. Minimum cuts in near-linear time. *Journal of the ACM*, 47:46–76, 2000.

[20] J S Provan and M O Ball. The complexity of counting cuts and of computing the probability that a graph is connected. *SIAM Journal of Computing*, 12:777–788, 1983.

[21] S Rai. A cutset approach to reliability evaluation in communication networks. *IEEE Transactions on Reliability*, 31:428–431, 1982.

[22] G Tel. *Introduction to distributed algorithms*. Cambridge University Press, Cambridge, UK, Second edition 2001.

[23] D E Whited, D R Shier, and J P Jarvis. Reliability computations for planar networks. *OSRA Journal of Computing*, 2(1):46–60, 1990.

A APPENDIX: Algorithm $\mathcal{DFS}$

Algorithm A.4 Algorithm $\mathcal{DFS}$ 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:

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

Algorithm A.5 Algorithm $\mathcal{DFS}$ 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:

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