Snap-Stabilizing Detection of Cutsets

Alain Cournier, Stéphane Devismes, and Vincent Villain

LaRIA, CNRS FRE 2733, Université de Picardie Jules Verne, Amiens, France
{cournier, devismes, villain}@laria.u-picardie.fr

Abstract. A *snap-stabilizing protocol*, starting from any configuration, always behaves according to its specification. Here, we present the first snap-stabilizing protocol for arbitrary rooted networks which detects if a set of nodes is a cutset. This protocol is based on the depth-first search (DFS) traversal and its properties. 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.

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 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* [1] 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 [3]. Thus, some heuristics have been designed for arbitrary graphs [4] and polynomial complete methods has developped for some particular class of graphs [5,6]. Several works have been also proposed in distributed (non self-stabilizing) systems [7,8]. To our best knowledge, nothing about cutsets has been proposed in self-stabilizing systems until now (so, 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

D.A. Bader et al. (Eds.): HiPC 2005, LNCS 3769, pp. 488–497, 2005.

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 [9]. 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.

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. We describe our protocol in Sections 4. In Section 5, we give a sketch of the proof of snap-stabilization of our protocol[1]. 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 p as the *label* q. We assume that the labels of p, stored in the set $Neig_p$, 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 we use, each processor executes the same program except *r*. We consider the local shared memory model of communication. The program of every processor consists 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 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

[1] See `http://www.laria.u-picardie.fr/$\sim$devismes/tr2005-04.pdf` for a complete proof.

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. 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* [10]. This definition captures the execution rate of the slowest processor. Given a computation 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 [11], 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 [12]. By definition, any execution of a wave protocol contains at least one initialization action. So, following [11], we propose a more simple definition of snap-stabilization holding for wave protocols.

Definition 1 (Snap-stabilization for Wave Protocols). *Let $\mathcal{T}$ be a task, and $\mathcal{SP}_\mathcal{T}$ a specification of $\mathcal{T}$. A wave protocol $\mathcal{P}$ is snap-stabilizing for $\mathcal{SP}_\mathcal{T}$ if and only if (i) at least one processor eventually executes a particular action of $\mathcal{P}$, and (ii) 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 call *path* of $G = (V,E)$ any sequence of processors $P = p_0,p_1,\ldots,p_k$ such that $\forall i, 1 \leq i \leq k, (p_{i-1}, p_i) \in E$. P is said *elementary* if $\forall i,j, 0 \leq i < j \leq k,\ p_i \neq p_j$. If

p_0,...,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. $G_S = (V_S,E_S)$ is the subgraph of $G = (V,E)$ induced by V_S if and only if $V_S \subseteq V$ and $E_S = E \cap (V_S)^2$. $G = (V,E)$ is said connected if and only if $\forall p, q \in V$ there exists a path between p and q in G. A connected componante of G is any connected subgraph of G maximal by inclusion. A connected undirected graph without any cycle is called a tree. The 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 of G rooted at *r*. The *height* of a node p in $Tree(r)$, noted $h(p)$, is 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 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 $Tree(r)$. We call *tree edges* the edges of E_T and *non-tree edges* the edges of $E \setminus E_T$. We call *non-tree neighbors* of p, nodes linked to p by a non-tree edge. $Tree(r)$ is a DFS spanning tree of $G = (V,E)$ if and only if $\forall\ (p,q) \in E$, $p \in Tree(q)$ or $q \in Tree(p)$.

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*. By definition, CS is a cutset of G if and only if there exists at least two connected componantes in G'. So, in the following, we particularize a node, called $CCRoot$, for each connected componantes in G'. Then, we deduce some results, the last one is a technical lemma which provide a way to locally detect if a node is a $CCRoot$.

Definition 2 (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 $Tree(r)$). In particular, by definition,* r *is a CCRoot if* r $\notin$ *CS.*

Lemma 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.*

Corollary 1. *There only exists one $CCRoot$ in each connected componante of G'.*

Theorem 1. *CS is a cutset if and only if there exists at least two $CCRoot$ in G'.*

Lemma 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: (i) $p \in C$, (ii) $\forall x \in Tree(p)$ 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 $\mathcal{DFS}$ refers to the snap-stabilizing depth-first search (DFS) protocol of [9]. 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 [13]. This general technique allows to simplify the design and proofs of Algorithm $\mathcal{CCRCDFS}$.

Definition 3 (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 [9] 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 ends at q (i.e., q holds the token and the token has visited all its neighbors), q backtracks the token to p.

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. We now describe the inputs, variables, and actions of Algorithm $\mathcal{CCRC}$.

Algorithm 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 $Next_p$.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 other processors using Algorithm $\mathcal{DFS}$.

Variables. In Algorithm $\mathcal{CCRC}$, each processor p maintains the following datas: (i) $Level_p$, Cnt_p, and $IsCutset_p$ for $p = r$; (ii) $Level_p$, $Back_p$, 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., the Uppermost Non-Tree Neighbor of $Tree(p)$ in C_p) as follows: **If** $p \in CS$, $UNNTC(p)$ = -1. **Otherwise**, p belongs to a connected componante 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 $q \in C_p$. From the definition of $UNNTC$ and Lemma 2, 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 2. *$\forall p \in V \setminus \{r\}$, p is a $CCRoot$ if and only if $p \notin CS$ and $h(p) = UNNTC(p)$.*

Algorithm 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:

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

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 3). 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 (Definition 2 allows to determine if r is a $CCRoot$ or not). Then, when the traversal locally ends 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 neighbors of p. In Algorithm $\mathcal{CCRC}$, the postfix action allows to:

- Compute $Back_p$ for $p \neq r$. Indeed, when the traversal ends at p, its neighbors have computed their height and its descendants have evaluated their $Back$ Variable.
- Update Cnt_p for $p \neq r$. As $Back_p$ and $Level_p$ are evaluated, by Theorem 2, p knows if it is a $CCRoot$ and, if necessary, it increments Cnt_p.
- Update $IsCutset_p$ for $p = r$. When the traversal ends at r, the traversal is entirely done. So, r knows the number of $CCRoots$ of the network and, using Theorem 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 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 Definition 2. Then, each time a processor $p \neq r$ receives the token for the first 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 ends 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 2, 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 1) and updates $IsCutset_r$ as well. Thus, from any initial configuration, after the end of a DFS traversal initiated by *r*, we obtain a configuration similar to the one shown in Figure 1. In this exemple, $CS = \{1, 6, 8\}$ and *r*, 2 are $CCRoots$. The root processor *r* is a $CCRoot$ because $r \notin CS$ (Definition 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 $CCRoots$ (here, equal to 2) and $IsCutset_r$ is set to true at the end of the traversal according to Theorem 1.

Algorithm 3 Algorithm ($\mathcal{CCRCDFS}$) CCRoots Counting and Depth-First Search

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

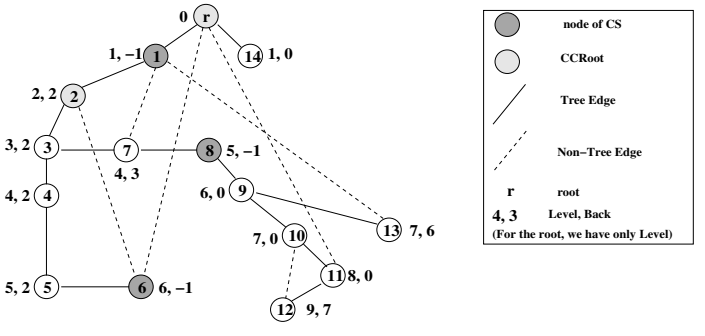

Fig. 1. State of the network after the end of a DFS traversal initiated by *r*

5 Sketch of Proof

In this section, we show 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, Algorithm $\mathcal{CCRC}$ has no impact on the behavior of Algorithm $\mathcal{DFS}$. From [9], 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. Since Algorithm $\mathcal{CCRC}$ cannot prevent Algorithm $\mathcal{DFS}$ to work as expected, 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 $\mathcal{DFS}$). So, if we focus on the traversal performed from *r*, if easy to verify that, after receiving the token for the first time, any $p \in V$ satisfies $Level_p = h(p)$ until the end of the traversal. Then, when the traversal ends at p, $Back_p = UNNTC(p)$ and, by Theorem 2, p is able to decide if it is a $CCRoot$ or not as explained in Section 4. Hence, at the end of a traversal initiated by r, *r* knows the number of $CCRoots$ and takes the right decision, i.e., $IsCutset_r = true$ if and only if CS is a cutset. Finally, in [9], Algorithm $\mathcal{DFS}$ is proven assuming an unfair daemon. Now, by Definition 3, Algorithm $\mathcal{CCRCDFS}$ works with the same number of steps than Algorithm $\mathcal{DFS}$ and it is snap-stabilizing under the unfair daemon.

Theorem 3. *Under an unfair daemon, Algorithm $\mathcal{CCRCDFS}$ is snap-stabilizing and detects if CS is a cutset.*

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 [9], we can deduce that a complete $\mathcal{CCRCDFS}$ computation is executed in $O(N^2)$ moves and in at most $6N - 1$ rounds.

Space Complexity. In Algorithms 1 and 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, we can deduce that the space requirement of Algorithm $\mathcal{CCRC}$ is $O(\log(N))$ bits per processor. From [9], we can conclude that the space requirement of Algorithm $\mathcal{CCRCDFS}$ is $O(N \times \log(N) + \log(\Delta))$ bits per processor (where Δ is an upper bound on the degree of the processors).

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, 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. [14,15]) in order to improve the memory requirement. Of course, in this case, the resulting protocol will be self-stabilizing only.

References

1. Dijkstra, E.: Self stabilizing systems in spite of distributed control. Communications of the Association of the Computing Machinery **17** (1974) 643–644
2. Bui, A., Datta, A., Petit, F., Villain, V.: State-optimal snap-stabilizing PIF in tree networks. In: Proceedings of the Fourth Workshop on Self-Stabilizing Systems, IEEE Computer Society Press (1999) 78–85
3. Provan, J.S., Ball, M.O.: The complexity of counting cuts and of computing the probability that a graph is connected. SIAM Journal of Computing **12** (1983) 777–788
4. Karger, D.R.: Minimum cuts in near-linear time. Journal of the ACM **47** (2000) 46–76
5. Ahmad, S.H.: Simple enumeration of minimal cutsets of acyclic directed graph. IEEE Transactions on Reliability **37** (1988) 484–487
6. Whited, D.E., Shier, D.R., Jarvis, J.P.: Reliability computations for planar networks. OSRA Journal of Computing **2(1)** (1990) 46–60
7. Fard, N.S., Lee, T.H.: Cutset enumeration of network systems with link and node failure. Reliability Engineering and System Safety **65** (1999) 141–146
8. Rai, S.: A cutset approach to reliability evaluation in communication networks. IEEE Transactions on Reliability **31** (1982) 428–431
9. Cournier, A., Devismes, S., Petit, F., Villain, V.: Snap-stabilizing depth-first search on arbitrary networks. In: OPODIS'04, International Conference On Principles Of Distributed Systems Proceedings. (2005) 267–282
10. Dolev, S., Israeli, A., Moran, S.: Uniform dynamic self-stabilizing leader election. IEEE Transactions on Parallel and Distributed Systems **8** (1997) 424–440
11. Cournier, A., Datta, A., Petit, F., Villain, V.: Enabling snap-stabilization. In: 23th International Conference on Distributed Computing Systems (ICDCS 2003). (2003) 12–19
12. Tel, G.: Introduction to distributed algorithms. Cambridge University Press (Second edition 2001)
13. Datta, A.K., Gurumurthy, S., Petit, F., Villain, V.: Self-stabilizing network orientation algorithms in arbitrary rooted networks. In: International Conference on Distributed Computing Systems. (2000) 576–583
14. Huang, S., Chen, N.: Self-stabilizing depth-first token circulation on networks. Distributed Computing **7** (1993) 61–66
15. Datta, A., Johnen, C., Petit, F., Villain, V.: Self-stabilizing depth-first token circulation in arbitrary rooted networks. Distributed Computing **13(4)** (2000) 207–218