A Silent Self-Stabilizing Algorithm for finding Cut-nodes and Bridges

Stéphane Devismes
LaRIA, CNRS FRE 2733
Université de Picardie Jules Verne
Amiens, France

July 16, 2005

Abstract

In this paper, we present a silent algorithm for finding cut-nodes and bridges in arbitrary rooted networks. This algorithm must be composed with an algorithm from Collin and Dolev. This latter algorithm is also silent and computes a Depth First Search (DFS) Spanning Tree of the network. The composition of these two algorithms is *self-stabilizing*: starting from any arbitrary configuration, it requires a finite number of steps to converge to a legitimate configuration and the system state remains legimate thereafter. The memory requirement of this composite algorithm is $O(n \times \log(\Delta) + \log(n))$ bits per processor where n is the number of processors and Δ is an upper bound on the degree of processors. Until now, this is the protocol with the lowest memory requirement solving this problem. Furthermore, our algorithm needs $O(n^2)$ moves to reach a terminal configuration once the DFS spanning tree is computed. This time complexity is equivalent to the best proposed solutions.

keywords : Distributed systems, self-stabilizing algorithms, undirected graphs, cut-node, bridge.

1 Introduction

Consider a connected undirected graph $G = (V, E)$ where V is the set of n nodes and E is the set of m edges. A node $p \in V$ is a *cut-node* (or an *articulation point*) of G if the removal of p disconnects G. In the same way, an edge $(p, q) \in E$ is a *bridge* if the removal of (p, q) disconnects G. When the graph represents a communication network then the existence of cut-nodes or bridges can become the potential cause for congestion in the network. Thus, from the fault tolerance point of view, the identification of cut-nodes and bridges of a network is crucial.

In this paper, we are interested into finding cut-nodes and bridges in distributed systems. Another desirable property for a distributed system is to withstand *transient failures*. The concept of *self-stabilization* [5] is the most general technique to design a system to tolerate 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 finite time. In such systems, a silent algorithm is an algorithm which, from any arbitrary initial configuration, reaches, in a finite number of steps, a terminal configuration where no processor can execute any action; this configuration must satisfy some properties for all possible executions.

1.1 Related Work

Some algorithms for finding cut-nodes and bridges have been proposed in the graph theory, e.g., by Paton [11] and Tarjan [13], the latter has a linear time complexity. This problem has also been investigated in the context of parallel and distributed computing [1, 10, 12]. In self-stabilizing systems, Collin and Dolev presented in [4] a silent algorithm whose output is a DFS spanning tree. Chaudhuri and Karaata present silent algorithms in [8, 9, 3, 2] for finding cut-nodes and bridges. Algorithms from [3, 2], which also use a silent algorithm for computing a DFS spanning tree, offer the best time complexity: they need $O(n^2)$ moves to stabilize once the DFS spanning tree is computed (instead of $O(n^2 \times m)$ for [8, 9]). However, for all these solutions, the memory requirement is $\Omega(m \times \log(m))$ bits per processor (without taking account of variables used for the spanning tree computing). Indeed, in these algorithms, every processor must maintain, locally, a list of edges.

1.2 Contributions

In this paper, we present a new silent self-stabilizing distributed algorithm for finding cut-nodes and bridges. This algorithm must be composed with the algorithm from [4]. Once the DFS spanning tree is computed by the algorihm from [4], our algorithm reaches a terminal configuration in $O(n^2)$ moves and $O(H)$ rounds where H is the height of the spanning tree. This time complexity corresponds to the best proposed solutions. Furthermore, the memory requirement of our algorithm is $O(\log(n))$ bits per processor. Thus, the memory requirement of the composition is $O(n \times \log(\Delta) + \log(n))$ bits per processor where Δ is an upper bound on the degree of processors. Until now, this is the protocol with the lowest memory requirement solving this problem.

1.3 Outline of the paper

In the next section (Section 2), we describe the distributed system and the model in which our algorithm is written. In the same section, we also state what it means for a composite algorithm to be self-stabilizing. In Section 3, we explain how the algorithm from [4] works. We present and prove our solution in Sections 4 and 5. In Section 6, we discuss about some complexity results. Finally, we conclude (Section 7).

2 Preliminaries

2.1 Distributed System

We consider a *distributed system* as an undirected connected graph $G = (V, E)$ where V is a set of *processors* ($|V| = n$) and E is the set of *bidirectional communication links*. We consider networks which are *asynchronous* and *rooted*, i.e., all processors, except the root, are *anonymous*. We denote the root processor by *r*. A communication link between two processors p and q will be denoted by (p, q). 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 α_p.We assume that $Neig_p$ is a constant, $Neig_p$ is shown as an input from the system. The local order α_p induced an enumeration of the links. Thus, for any link $e = (p, q)$, let $\alpha_p(q)$ ($\alpha_q(p)$, respectively) be the link index of e according to α_p (α_q, respectively). We assume that for every processor p and any edge (p, q), p knows the value of $\alpha_q(p)$ (For more informations see [6]). We recall that, in most cases, $\alpha_p(q) \neq \alpha_q(p)$. Finally, for each link (p, q), $\alpha_p(q)$ and $\alpha_q(p)$ are also constants and shown as inputs from the system.

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

2.2 Computational Model

In the computation model that we use each processor executes the same program except *r*. We consider the local shared memory model of communication. The program of every processor consists of a set of *shared variables* (henceforth, referred to as variables) and a finite set of actions. A processor can only write to its own variables, and read its own variables and variables owned by the neighboring processors. Each action is of the following form:

$$< label > :: < guard > \rightarrow < statement > .$$

The guard of an action in the program of p is a boolean expression involving the variables of p and its neighbors. The statement of an action of p updates one or more variables of p. An action can be executed only if its guard is satisfied. We assume that the actions are atomically executed, meaning, the evaluation of a guard and the execution of the corresponding statement of an action, if executed, are done in one atomic step.

The *state* of a processor is defined by the value of its variables. The *state* of a system is the product of the states of all processors ($\in V$). We will refer to the state of a processor and system as a (*local*) *state* and (*global*) *configuration*, respectively. Let $\mathcal{C}$, the set of all possible configurations of the system. An action A is said to be enabled in $\gamma \in \mathcal{C}$ at p if the guard of A is true at p in γ. A processor p is said to be *enabled* in γ ($\gamma \in \mathcal{C}$) if there exists an enabled action A in the program of p in γ. Let a distributed protocol $\mathcal{P}$ be a collection of binary transition relations denoted by $\mapsto$, on $\mathcal{C}$. A *computation* of a protocol $\mathcal{P}$ is a *maximal* sequence of configurations $e = (\gamma_0, \gamma_1, ..., \gamma_i, \gamma_{i+1}, ...)$, such that for $i \geq 0$, $\gamma_i \mapsto \gamma_{i+1}$ (called a *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 $\mathcal{P}$ in system S is denoted as $\mathcal{E}$. We consider that any processor p executed a *disable 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 disable action represents the following situation: at least one neighbor of p changes its state between γ_i and γ_{i+1}, and this change effectively made the guard of all actions of p false).

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

In order to compute the time complexity, we use the definition of *round* [7]. 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 disable action) of every enabled processor in the first configuration. Let e'' be the suffix of e such that $e = e'e''$. Then *second round* of e is the first round of e'', and so on.

2.3 Self-Stabilizing System

Let PRE be a predicate defined over $\mathcal{C}$. The protocol $\mathcal{P}$ running on the distributed system $\mathcal{S}$ is said to be *self-stabilizing* with respect to PRE if it satisfies:

- **Closure:** If a configuration γ satisfies PRE, then any configuration that is reachable from γ using $\mathcal{P}$ also satisfies PRE.

- **Convergence:** Starting from an arbitrary configuration, the distributed system $\mathcal{S}$ is guaranteed to reach a configuration satisfying PRE in a finite number of steps of $\mathcal{P}$.

Configurations satisfying PRE are said to be *legitimate*. Similary, a configuration that does not satisfy PRE is referred to as an *illegitimate state*. To show that an algorithm is self-stabilizing with respect to PRE, we need to show the satisfiability of both closure and convergence conditions. In the case of silent algorithms, we have just to show the convergence to a terminal configuration that satisfies PRE. After, because this configuration is terminal, the closure is trivially satisfied.

2.4 Definitions and Notations

Definition 1 (Path) *The sequence of nodes p_1, ..., p_k is a path of G if and only if $\forall\, i \in$ [1, ..., k−1], (p_i,p_{i+1}) $\in$ E (the set of edges of G).*

Definition 2 (Length of a Path) *The length of a path P, noted $length(P)$, is the number of edges which compose P.*

Definition 3 (Partial Graph) *The graph G_A = (V, A) is a partial graph of G = (V, E) if and only if A $\subseteq$ E.*

Definition 4 (Subgraph) *The subgraph of G = (V, E) induced by S (S $\subseteq$ V) is the graph G_S=(S, E_S) such that E_S=E $\cap$ S^2.*

Definition 5 (Connected Graph) *An undirected graph G_C is connected if and only if, for each pair of distinct nodes (p, q), there exists a path in G_C between p and q.*

Definition 6 (Spanning Tree) *A graph T = (V, E_T) is a spanning tree of G if and only if T is a partial connected graph of G where $|E_T| = n - 1$.*

In a rooted spanning tree $T(r) = (V, E')$, we distinguish a node *r* called "root". We define the *height* of a node p in $T(r)$ (noted $h(p)$) as the length of the simple path (loopless) from *r* to p in $T(r)$. $H = \max_{p \in T(r)}\{h(p)\}$ represents the height of $T(r)$. For a node $p \neq r$, a node $q \in V$ is said to be the *parent* of p in $T(r)$ (noted $parent(p)$) if and only if q is the neighbor of p such that $h(p) = h(q) + 1$, conversely, p is said to be the *child* of q in $T(r)$. $C(p)$ denotes the set of children of a node p in $T(r)$. A node p_1 is said to be an ancestor of another node p_k in $T(r)$ (with $k > 1$) if there exists a sequence of nodes p_1, ..., p_k such that $\forall\, p_j$, with $1 \leq j < k$, p_j is the parent of p_{j+1} in $T(r)$, conversely p_k is said to be a *descendant* of p_1. We will note $T(p)$ the subtree of $T(r)$ rooted at p ($\in V$), i.e., the subgraph of $T(r)$ induced by p and its descendants in $T(r)$. Finally, we will call *tree edges*, the edges of E' and *non-tree edges*, the edges of $E \setminus E'$. We will call *non-tree neighbors* of p, nodes linked to p by a non-tree edge.

Definition 7 (DFS Spanning Tree[2]) *T(r) is a DFS spanning tree of G if and only if T(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 is either an ancestor or a descendant of q in T(r).*

From Definition 7, we can deduce this useful property about the DFS Spanning Tree.

Property 1 *Let $p \in V$. Let T(r) be a DFS Spanning Tree of G (rooted at r). $\forall q \in T(p)$, every non-tree neighbor of q is either an ancestor or a descendant of p in T(r).*

Now, we give the formal definitions of cut-nodes and bridges.

[2]This definition holds for undirected graphs only.

Definition 8 (Cut-node) *A node $p \in V$ is a cut-node (or an articulation point) of G if and only if the subgraph of G induced by $V \setminus \{p\}$ is disconnected.*

Definition 9 (Bridge) *An edge $(p, q) \in E$ is a bridge of G if and only if the partial graph $G' = (V, E \setminus \{(p,q)\})$ is disconnected.*

2.5 Protocol Composition

These following definitions and theorem (from [14]) explain a way to prove that a composition of algorithms is self-stabilizing.

Definition 10 (Collateral Composition) *Let S_1 and S_2 be programs such that no variables written by S_2 appears in S_1. The collateral composition of S_1 and S_2, denoted $S_2 \circ S_1$, is the program that has all the variables and all the actions of S_1 and S_2.*

Let L_1 and L_2 be predicate over the variables of S_1 and S_2, respectively. In the composite algorithm, L_1 will be established by S_1, and subsequently, L_2 will be established by S_2. We now define a *fair* composition with respect to both programs, and define what it means for a composite algorithm to be self-stabilizing.

Definition 11 (Fair Execution) *An execution e of $S_1 \circ S_2$ is fair with respect to S_i ($i \in \{1,2\}$) if one of these conditions holds:*

1. *e is finite;*
2. *e contains infinitely steps of S_i, or contains an infinite suffix in which no action of S_i is enabled.*

Definition 12 (Fair Composition) *The composition $S_2 \circ S_1$ is fair with respect to S_i ($i \in \{1,2\}$) if every execution of $S_2 \circ S_1$ is fair with respect to S_i.*

Theorem 1 *$S_2 \circ S_1$ stabilizes to L_2 if the following four conditions hold:*

1. *Program S_1 stabilizes to L_1;*
2. *Program S_2 stabilizes to L_2 if L_1 holds;*
3. *Program S_1 does not change variables read by S_2 once L_1 holds;*
4. *The composition is fair with respect to both S_1 and S_2.*

3 The DFS algorithm of Collin and Dolev

In this section, we present the algorithm of Collin and Dolev, referred to as Algorithm $\mathcal{DFS}$, from [4]. This is a silent algorithm which computed a DFS spanning tree in a distributed fashion. Furthermore, it stabilizes in a finite number of moves.

First, to compute the DFS spanning tree, this algorithm uses the notion of *first path*.

Definition 13 (First Path) *For each simple (loopless) path from the root P = (p_1=r), ..., p_i, ...,p_k of G, we associate a word n_0, ...,n_i, n_{k-1} (noted $word(P)$) where n_0 = $\perp$ and, $\forall$ i $\in$ [1, ...k $-$ 1], p_i is linked to p_{i+1} by the edge of index n_i on p_i (i.e., $n_i = \alpha_{p_i}(p_{i+1})$). Then, we define a lexicographical order $\prec_{lex}$ over these words where $\perp$ is the minimal character. For each processor p, we define the set of all simple paths from the root r to p. The path of this set with the minimal word by $\prec_{lex}$ is called the first path of p (noted $fp(p)$).*

Using this notion, we can specify the *first* DFS spanning tree.

Specification 1 (First DFS Spanning Tree) *T(r) is the first DFS spanning tree of G if and only if T(r) is a spanning tree of G (rooted at r) and $\forall$ $p \in V$ the path from r to p in T(r) is the first path of p in G.*

From Specification 1, Collin and Dolev have implemented Algorithm $\mathcal{DFS}$. Initially, in [4], Algorithm $\mathcal{DFS}$ has been written in the register model. However, this model is close to the state model. Thus, we design Algorithm $\mathcal{DFS}$ in the state model (see Algorithms 1 and 2). Informally, Algorithm $\mathcal{DFS}$ works as follows: the memory of any processor p consists of a *path* field denoted by $Path_p$. The root *r* has its constant $Path_r$ equal to $\perp$. Any other processor p repeatedly reads the variables of its neighbors. The path $Path_q$, read by p from the neighbor q, derives a path for p simply by concatening $Path_q$ with $\alpha_q(p)$ (noted $Path_q \oplus \alpha_q(p)$). We recall that the value of $\alpha_q(p)$ is known to p. Then, p chooses its path to be the minimal path among the paths derived from its neighbors' paths.

For any processor p, Path$_p$ contains a sequence of at most N items ($N \geq n$) where an item is $\perp$ or an edge index. Indeed, $Path_p$ may describe the longuest path that is possible in G. Thus, the notation right$_k(w)$ refers to the sequence of the k least significant items of w.

In a terminal configuration, Algorithm $\mathcal{DFS}$ satisfies this following predicate:

$$PRE_{DFS}.\ \forall p \in V,\ Path_p \text{ contains } word(fp(p)).$$

Thus, from [4], we can claim the following theorem:

Theorem 2 *Algorithm $\mathcal{DFS}$ stabilizes to PRE_{DFS}.*

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

Input: $Neig_p$: set of neighbors (locally ordered);
Constant: $Path_p$=$\perp$;

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

Input: $Neig_p$: set of neighbors (locally ordered);
Constant: $N \geq n$;
Variable: $Path_p$: list of, at most, N ordered items $\in \{\perp\} \cup \mathbb{N}$;
Macro:
$Read\_path_p = \bigcup_{q \in Neig_p} \{\text{right}_N(Path_q \oplus \alpha_q(p))\}$;
Predicates:
$Update\_path(p) \equiv (Path_p \neq \min_{\prec_{lex}}(Read\_path_p))$
Action:
$Compute\_path(p)$ :: $Update\_path(p) \rightarrow Path_p := \min_{\prec_{lex}}(Read\_path_p)$;

Remark 1 (Parent) *In a terminal configuration, the parent of p is the only processor q which satisfies: $q \in Neig_p \wedge Path_p = Path_q \oplus \alpha_q(p)$.*

Remark 2 (Child) *In a terminal configuration, $q \in C(p)$ (i.e., q is a child of p) if and only if $q \in Neig_p \wedge Path_q = Path_p \oplus \alpha_p(q)$.*

Remark 3 (Height) *In a terminal configuration, $\forall\ p \in V$, $Path_p$ contains $\perp$ followed by the sequence of edge indices from r to p in the DFS spanning tree. Thus, h(p) (height of p) in the DFS spanning tree is equal to $|Path_p| - 1$.*

From now, we will note $T(r) = (V, E')$ the $first\ DFS$ spanning tree rooted at r computed by Algorithm $\mathcal{DFS}$ and H the height of $T(r)$.

4 Algorithm

In this section, we present a silent algorithm called Algorithm $\mathcal{UNNS}$[3](see Algorithms 3 and 4). This algorithm must be composed with Algorithm $\mathcal{DFS}$ (shown in the above section).

4.1 Approach

To implement our algorithm, we use two theorems established by Tarjan in [13]:

Theorem 3 *r (root of G) is a cut-node if and only if $|C(r)| \geq 2$.*

Theorem 4 *$\forall p \in V \setminus \{r\}$, p is a cut-node if and only if there exists a node $q \in C(p)$ for which no node in T(q) is linked by a non-tree edge to an ancestor of p in T(r).*

These theorems can be deduced from Definition 7. Figure 1 depicts a DFS spanning tree $T(r)$ of a connected undirected graph G. The root r has exactly two children in $T(r)$. We can remark that the removal of r would disconnect the graph into two connected components: the subgraph induced by 2 and its descendants and the subgraph induced by 1 and its descendants, therefore r is a cut-node. In the same way, the removal of the node 3 would disconnect the subgraph induced by 7 and its descendants from the rest of the graph because no node of $T(7)$ is linked by a non-tree edge to an ancestor of 3.

[3]means Uppermost Non-tree Neighbor of each Subtree

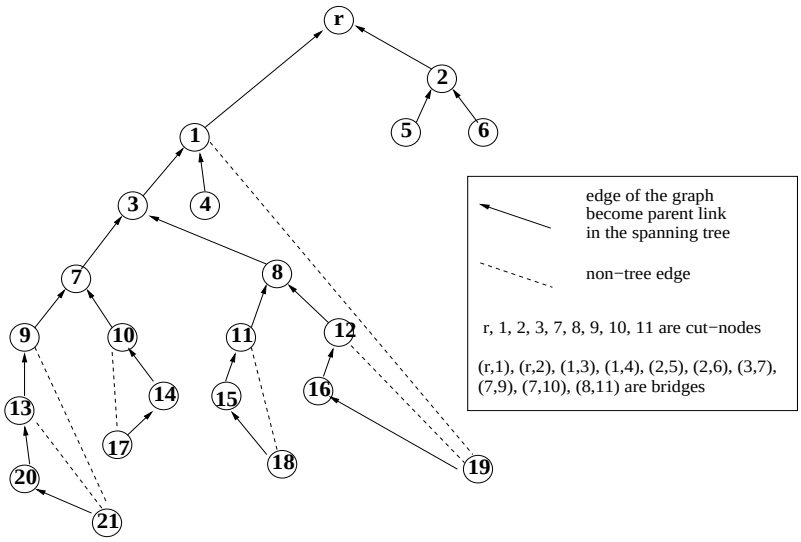

Figure 1: DFS spanning tree in a connected undirected graph.

The following remark shows that cut-nodes and bridges have close relations.

Remark 4 *If an edge (p,q) $\in E$ is a bridge of G then each of its both incident nodes is either a cut-node or a pendent-node (i.e., a node of degree one).*

Algorithm 3 Algorithm $\mathcal{UNNS}$ for $p = r$

Input:
$Path_p$: constant from $\mathcal{DFS}$;
$Neig_p$: set of neighbors (locally ordered);
Macros:
$Height_p = |Path_p| - 1$;
$Children_p = \{q \in Neig_p :: (Path_p \oplus \alpha_p(q)) = Path_q\}$;

Algorithm 4 Algorithm $\mathcal{UNNS}$ for $p \neq r$

Input:
N: constant from $\mathcal{DFS}$;
$Path_p$: list of, at most, N ordered items $\in \{\perp\} \cup \mathbb{N}$ from $\mathcal{DFS}$;
$Neig_p$: set of neighbors (locally ordered);
Variable: $Back_p \in [0, ..., N-1]$;
Macro:
$Height_p = |Path_p| - 1$;
$Par_p = q$ **if** $(\exists!\ q \in Neig_p :: (Path_q \oplus \alpha_q(p)) = Path_p)$, $\perp$ **otherwise**;
$Children_p = \{q \in Neig_p :: (Path_p \oplus \alpha_p(q)) = Path_q\}$;
$Children\_back_p = \{x \in [0, ..., N-1] :: (\exists q \in Children_p :: Back_q = x)\}$;
$Nontree\_height_p = \{x \in [0, ..., N-1] :: (\exists q \in Neig_p :: q \notin Children_p \wedge Par_p \neq q \wedge Height_q = x)\}$;
Predicate:
$Update\_back(p) \equiv (Back_p \neq \min(Children\_back_p \cup Nontree\_height_p \cup \{Height_p\}))$
Actions:
$Compute\_back(p)$ :: $Update\_back(p) \rightarrow Back_p := \min(Children\_back_p \cup Nontree\_height_p \cup \{Height_p\})$;

From Theorem 4 and Remark 4, we know that the notion of non-tree neighbor is fondamental to determine cut-nodes and then bridges. Thus, for each processor $p \in V \setminus \{r\}$, Algorithm $\mathcal{UNNS}$ computes $u(p)$ = $\min_{x \in T(p)}(\{\ h(y) :: (x,y) \in E - E'\} \cup \{h(x)\})$, where E' is the edge set of $T(r)$. Informally, $u(p)$ corresponds to the minimal value among the height of each node of $T(p)$ and the height of the non-tree neighbors of $T(p)$, if they exist (for instance, in figure 1, Node 1 is a non-tree neighbor of $T(8)$). In our algorithm, we use $Path_p$ as an input from Algorithm $\mathcal{DFS}$ to know the parent of each node in $T(r)$ (see Macro Par_p and Remark 1) as well as their children (see Macro $Children_p$ and Remark 2) and $h(p)$ (see macro $Height_p$ and Remark 3).

In Section 5, we will prove that, for all possible executions, Algorithm $\mathcal{UNNS}$ reaches a terminal configuration and satifies the following predicate in this configuration:

PRE_{UNNS}. For every node $p \in V \setminus \{r\}$, $Back_p$ is equal to $u(p)$.

4.2 Detection of Cut-nodes and Bridges

Now, we show that if $\forall p \in V$, we know $h(p)$, $u(p)$, $parent(p)$ and $C(p)$ then we can easily detect all the cut-nodes and the bridges of G.

Proposition 1 *A node $p \in V$ is a cut-node if and only if p satisfies one of the following two conditions:*

1. *($p =$ r) $\wedge$ ($|C(p)| \geq 2$);*
2. *($p \neq$ r) $\wedge$ ($\exists q \in C(p) :: u(q) \geq h(p)$).*

Proof. First, from Theorem 3, we can trivially deduce that 1. is equivalent to the proposition "r is a cut-node". Then, for 2., by definition, $u(p)$ is equal to the lowest height among the height of each node of $T(p)$ and the height of the non-tree neighbors of $T(p)$. Now, assume that a node $p \in V \setminus \{r\}$ is a cut-node. From Theorem 4, we know that there exists a child q of p in $T(r)$ such that no node in $T(q)$ are linked to an ancestor of p by a non-tree edge, i.e., each non-tree neighbor of $T(q)$ has a height in $T(r)$ greater or equal to the height of p (From Property 1, each non-tree neighbor of $T(q)$ is a descendant of p). Thus, $u(q) \geq h(p)$. Hence, if p ($p \neq r$) is a cut-node then 2. is true. With the same arguments we can trivially deduce the reciprocal. □

Proposition 2 *$\forall\, p \in V \setminus \{r\}$, Edge $(p, parent(p))$ is a bridge if and only if $u(p) = h(p)$.*

Proof. First, we can assert that a bridge $(p, q) \in E'$ (the set of edge of $T(r)$) because, by definition, the bridge (p, q) is the only way to go from p to q in G (respectively from q to p).

Now, assume that the edge $(p, parent(p))$ is a bridge and $u(p) \neq h(p)$. Then, $u(p)$ is strictly lower than $h(p)$ because $u(p) = \min_{x \in T(p)}(\{\ h(y) :: (x,y) \in E - E'\} \cup \{h(x)\})$. Now, if $u(p)$ is strictly lower than $h(p)$ that means that there exists a non-tree edge between an ancestor of p and a node of $T(p)$ (by definition of $u(p)$ and Property 1). Thus, the removal of $(p, parent(p))$ does not disconnect G. Contradiction.

Finally, assume that the edge $(p, parent(p))$ is not a bridge and $u(p) = h(p)$. In this case, there exists, at least, one cycle in G including the edge $(p, parent(p))$. As $T(r)$ is the $first\ DFS$ spanning tree, this cycle is composed with tree edges and one non-tree edge. This non-tree edge links a node of $T(p)$ to an ancestor of p. Thus, $u(p) < h(p)$ (by definition of $u(p)$). Contradiction. □

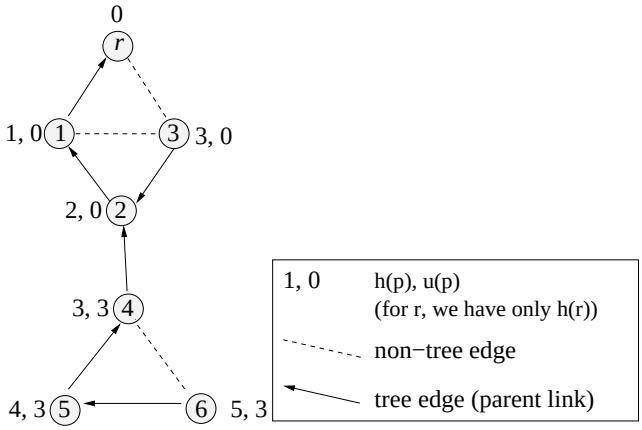

Figure 2: Example on an arbitrary network.

4.3 Example

4.3.1 Cut-nodes

In Figure 2, *r* has only one child so *r* is not a cut-node. Node 4 satisfies $u(4) \geq h(2)$ so its parent (Node 2) is a cut-node, in the same way, Node 4 is a cut-node. On the other hand, no child p of Node 1 satisfies $u(p) \geq h(1)$ so Node 1 is not a cut-node.

4.3.2 Bridges

In Figure 2, Node 4 satisfies $u(4) = h(4)$ so Edge (4,2) is a bridge. On the other hand, node 5 satisfies $u(5)$ = 3 and $h(5) = 4$ so Edge (5,4) is not a bridge.

5 Proof of Correctness

In this section, we prove that the composite algorithm $\mathcal{UNNS} \circ \mathcal{DFS}$ is self-stabilizing for the predicate PRE_{UNNS}. Thus, we assume, first, that the daemon is weakly fair. Then, we prove that Algorithm $\mathcal{UNNS} \circ \mathcal{DFS}$ stabilizes even if the daemon is unfair.

Lemma 1 *Every execution of Algorithm $\mathcal{UNNS} \circ \mathcal{DFS}$ has a finite number of moves.*

Proof. From [4], we know that Algorithm $\mathcal{DFS}$ has a finite number of moves. Thus, we have just to prove that, for any configuration of Algorithm $\mathcal{DFS}$, Algorithm $\mathcal{UNNS}$ can execute a finite number of steps only.

By hypothesis, we consider that Algorithm $\mathcal{DFS}$ does not make any action. Then, we can assume that $\forall$ $p \in V$ the values returned by the macros Par_p and $Children_p$ are set. Let $G_c = (V, E_c)$ be the partial graph of G where $E_c = \{(p,q) \in E :: Par_p = q\}$ for an arbitrary configuration of Algorithm $\mathcal{DFS}$. Thus, G_c is the partial graph of G computed by Algorithm $\mathcal{DFS}$. We focus on G_c because the updating of $Back_p$ for a processor p can only *cause* the execution of $Compute\_back$ of the processor pointed by the macro Par_p, if it exists (see Predicate $Update\_back$).
In any "system" G_c, we are interested in the causes of the moves. Thus, a move can be caused by:

- an initial configuration;

- a change in the state of a neighbor.

We call an *initial* move a move caused by an initial configuration. By definition, the number of initial moves is bounded (by $n-1$, since *r* has no action in Algorithm $\mathcal{UNNS}$). So, we have to show that the number of the other moves is also bounded.
For each connected component $CC_i = (V_i, E_i)$ of G_c two cases are possible:

1. CC_i is a tree: then, in the worst case, each processor $p \in V_i \setminus \{r\}$ can execute $Compute\_back(p)$ as an initial move (the program of *r* contains no action in Algorithm $\mathcal{UNNS}$). Moreover, the updating of $Back_p$ can only cause the execution of $Compute\_back$ of the processor pointed by the macro Par_p, if it exists (see Predicate $Update\_back$). Now, we can remark that, for the nodes of V_i, the *cause relationship* of the action $Compute\_back$ is *acyclic* (indeed, the $Back$s' updatings go up in $T(r)$ following the Par variables) and the source of all chains of $Back_p$ updating are always initial moves. Thus, the length each chain of $Back_p$ updating in CC_i is bounded by the height of CC_i. Thus, if CC_i is a tree, the number of $Compute\_back$ moves in CC_i is finite.

2. CC_i is not a tree: because for each processor $p \in V_i$, Par_p pointed, at most, one processor and CC_i is connected, CC_i contains one cycle only and CC_i is not rooted (i.e., there does not exist a processor $q \in V_i$ such that $Par_q = \perp$). We call this cycle C and *c* the number of its processors.

   In the worst case, each processor $p \in V_i$ can execute $Compute\_back(p)$ as an initial move.

   Consider a enabled processor $q \in V_i$ which does not belong to C. The execution of $Compute\_back(q)$ can only cause the execution of the action $Compute\_back$ of the processor pointed by the macro Par_q (see Predicate $Update\_back$) and so on. In the worst case, these executions follow a path from q to a processor of C.

   Now, consider the case where the action $Compute\_back$ of a processor q in C is or becomes enabled. The execution of $Compute\_back(q)$ can only activate the processor pointed by Par_q (see Predicate $Update\_back$) and follows C. However, these executions propagate the same value in the $Back$ variables. Thus, in the worst case, the processor q can, at most, induce $c-1$ executions of $Compute\_back$ actions (one for each other node of C). Indeed, a processor becomes enabled in order to assign a new value to its $Back$ variable only. Thus, even if CC_i is not a tree, the number of $Compute\_back$ moves in CC_i is also finite.

□

Let $p \in V$. Let f be the leaf with the upper height in $T(p)$.We define $d(p)$ as follows:

$$d(p) = h(f) - h(p).$$

Thus, $d(p)$ represents the distance between p and the upper height leaf of its induced subtree.

Lemma 2 *Algorithm $\mathcal{UNNS}$ stabilizes to PRE_{UNNS} if PRE_{DFS} holds.*

Proof. We begin the proof with some claims. First, from [4] and Lemma 1, we know that $\mathcal{UNNS} \circ \mathcal{DFS}$ reaches a terminal configuration in a finite number of moves and, in this configuration, PRE_{DFS} holds. Assuming PRE_{DFS} holds, all the edges (p, Par_p) such that $p \in V \setminus \{r\}$ shape the $first\ DFS$ spanning tree $T(r)$, the macro $Children_p$ equals $C(p)$, and the macro $Height_p$ returns $h(p)$ (see Remarks 1, 2 and 3). Since the system reaches a configuration where no action is enabled, the predicate $Update\_back(p)$ is false for each $p \in V \setminus \{r\}$.

Thus, we have to prove that, in this configuration, $\forall\, p \in V \backslash \{r\}$ Back$_p = u(p)$. We prove that by induction on $d(x)$ in $T(r)$.

Let $L(r)$ the set of leaves of $T(r)$. In the terminal configuration, $\forall\ f \in L(r)$ ($d(f)$=0), because $Children\_back_f = \emptyset$, $Back_f = \min(Nontree\_height_f \cup \{Height_f\})$. Now, $Nontree\_height_f = \bigcup_{y \in Neig_f \backslash \{Par_f\}} \{h(y)\}$, $C(f) = \emptyset$, and, $T(f) = \{f\}$. Then, $Back_f = \min_{x \in T(f)}(\{\ h(y) :: (x, y) \in E - E'\} \cup \{h(x)\}) = u(f)$.

Now, assume that for each node $p \in V \setminus \{r\}$, such that $d(p) \leq k$ ($k \geq 0$), we have $Back_p = u(p)$.

Consider the nodes $q \in V \setminus \{r\}$, such that $d(q) = k + 1$. In the terminal configuration, $Back_q = \min(Children\_back_q \cup Nontree\_height_q \cup \{Height_q\})$. By induction assumption, $Children\_back_q = \bigcup_{z \in C(q)} \{u(z)\} = \bigcup_{z \in C(q)} \{\min_{x \in T(z)}(\{h(y) :: (x, y) \in E - E'\} \cup \{h(x)\})\}$. $Nontree\_height_q = (\bigcup_{y \in Neig_q \backslash C(q) \backslash \{Par_q\}} \{h(y)\}) = \{h(y) :: (q, y) \in E - E'\}$. Thus, $Back_q = \min(\bigcup_{z \in C(q)} \{\min_{x \in T(z)}(\{h(y) :: (x, y) \in E - E'\} \cup \{h(x)\})\} \cup \{h(y) :: (q, y) \in E - E'\} \cup \{h(q)\}) = \min(\bigcup_{z \in C(q)} (\bigcup_{x \in T(z)}(\{h(y) :: (x, y) \in E - E'\} \cup \{h(x)\})) \cup \{h(y) :: (q, y) \in E - E'\} \cup \{h(q)\})$. Now, $T(q) = (\bigcup_{z \in C(q)} \{T(z)\}) \cup \{q\}$. Hence, $Back_q = \min_{x \in T(q)}(\{h(y) :: (x, y) \in E - E'\} \cup \{h(x)\}) = u(q)$. Thus, $\forall\, q \in V \setminus \{r\}$, such that $d(q) = k + 1$, $Back_q = u(q)$. Hence, this property is true for each processor p such $d(p) \leq k + 1$. With $k = H$ - 1 (because $Compute\_back$ does not exist in the program of r) the lemma holds. □

Theorem 5 *Algorithm $\mathcal{UNNS} \circ \mathcal{DFS}$ stabilizes to PRE_{UNNS}.*

Proof. From the following four observations and Theorem 1, the result holds.

1. From [4], we know that Algorithm $\mathcal{DFS}$ stabilizes to PRE_{DFS}.
2. By Lemma 2, Algorithm $\mathcal{UNNS}$ stabilizes to PRE_{UNNS} if PRE_{DFS} holds.
3. Because Algorithm $\mathcal{DFS}$ is silent, trivially, we can claim that Algorithm $\mathcal{DFS}$ does not change variables read by PRE_{UNNS} once PRE_{DFS} holds.
4. From Lemma 1, we know that every execution of $\mathcal{UNNS} \circ \mathcal{DFS}$ is finite. Thus, the composition $\mathcal{UNNS} \circ \mathcal{DFS}$ is fair with respect to both PRE_{DFS} and PRE_{UNNS} (see Definition 11).

□

We presented Algorithm $\mathcal{UNNS} \circ \mathcal{DFS}$ under the weak fairness assumption. The following theorem claims that Algorithm $\mathcal{UNNS} \circ \mathcal{DFS}$ is also correct without any fairness assumption. In fact, we prove that any execution of Algorithm $\mathcal{UNNS} \circ \mathcal{DFS}$ has a bounded number of moves (see Lemma 1). Therefore, an unfair daemon cannot forever prevent any enabled processor to execute an action.

Theorem 6 *Algorithm $\mathcal{UNNS} \circ \mathcal{DFS}$ stabilizes even if the daemon is unfair.*

Finally, from Theorems 2 and 5, Propositions 1 and 2, and Remark 1, we can claim the following theorem.

Theorem 7 *Algorithm $\mathcal{UNNS} \circ \mathcal{DFS}$ is self-stabilizing and detects all cut-nodes and bridges of G.*

Corollary 1 *After Algorithm $\mathcal{UNNS} \circ \mathcal{DFS}$ terminates, a node $p \in V$ is a cut-node if and only if p satisfies one of the two following conditions:*

1. *($p =$ r) $\wedge$ ($|\{q \in Children_p\ \}| \geq 2$)*
2. *($p \neq$ r) $\wedge$ ($\exists q \in Children_p :: (Back_q \geq Height_p)$).*

Corollary 2 *After Algorithm $\mathcal{UNNS} \circ \mathcal{DFS}$ terminates, $\forall\ p \in V\backslash\{r\}$, ($p$,$Par_p$) is a bridge if and only if $Back_p = Height_p$.*

6 Complexity

In order to compare our algorithm with solutions proposed in the literature, we compute the time complexity of Algorithm $\mathcal{UNNS}$ after Algorithm $\mathcal{DFS}$ terminates. Thus, in this section, we assume the presence of the $first\ DFS$ spanning tree $T(r)$. Moreover, we presented the space complexity of our solution.

6.1 Time Complexity

Theorem 8 *Algorithm $\mathcal{UNNS}$ needs $O(H)$ rounds to reach a terminal configuration after Algorithm $\mathcal{DFS}$ terminates.*

Proof. Since we assume that Algorithm $\mathcal{DFS}$ is terminated, the $first\ DFS$ tree $T(r)$ has been computed: $Path_p$, $Children_p$, and $Height_p$ are constant ($\forall\ p \in V$) and Par_p too ($\forall\ p \in V \setminus \{r\}$). Hence, $\forall\ p \in V \setminus \{r\}$, if $Compute\_back(p)$ is disabled then it can become enabled if and only if at least one of its children q in $T(r)$ has modified its variable $Back_q$ (see Predicate $Update\_back(p)$).

We prove this lemma by induction on $d(x)$ (distance from x to its farther leaf) in $T(r)$.

Let $f \in L(r)$, the set of leaves of $T(r)$ ($d(f) = 0$), Action $Compute\_back(f)$ depends on $Height$ variables only, so after one round, $Compute\_back(f)$ is disabled forever.

Now, assume that $\forall\ p \in V \setminus \{r\}$, such that $d(p) \leq k$ ($k \geq 0$), we have $Compute\_back(p)$ disabled forever and $Back_p$ is now constant after, at most, $d(p) + 1$ rounds.

For each $q \in V \setminus \{r\}$, $Update\_back(q)$ uses only $Back$ values of its children and $Height$ variables (which are constant). So, during the round $k+2$, each $q \in V\setminus\{r\}$ such that $d(q)=k+1$ reads $Back$ and $Height$ values which are constant from now. If $Compute\_back(q)$ is disabled, it will remain forever. Otherwise $Compute\_back(q)$ is continuously enabled until q executes it. So, after the round $k+2$, $Compute\_back(q)$ is disabled forever. At the end of the round H ($\forall\ q \in V \setminus \{r\}$, $d(q)<H$) no $Compute\_back$ action is enabled in the system. □

Theorem 9 *Algorithm $\mathcal{UNNS}$ needs $O(n^2)$ moves to reach a terminal configuration after Algorithm $\mathcal{DFS}$ terminates.*

Proof. In order to prove this time complexity, we use the notions of causes of the moves again. We recall that a move can be caused by:

- an initial configuration;
- a change in the state of a neighbor.

We call an *initial* move: a move caused by an initial configuration. As we assume Algorithm $\mathcal{DFS}$ is terminated, we can consider that, $\forall\ p$, $Path_p$ is set and, thus, the values returned by the macros Par_p (if $p \neq r$), $Children_p$, and $Height_p$ are also set. Hence, there exists a DFS spanning tree rooted at r, $T(r) = (V, E')$ such that $E' = \{(p, Par_p)::\ p \in V \setminus \{r\}\}$. In the worst case, each processor $p \in V \setminus \{r\}$ can execute $Compute\_back(p)$ as an initial move. So, the total number of execution of initial $Compute\_back$ is in $O(n)$ moves. Then, the updating of the variable $Back_p$ of a processor p can only *cause* the execution of $Compute\_back$ of its parent in $T(r)$, i.e., Par_p (see Predicate $Update\_back$). Moreover, we can remark that the *cause relationship* of the action $Compute\_back$ is *acyclic* (indeed, the $Back$s' updatings go up in $T(r)$

following the Par variables) and the source of all chains of $Back$s' updating is always an initial move. Thus, the length of each chain of $Back$s' updating is bounded by H, the height of $T(r)$ ($H \leq n$). Hence, the number of executions of $Compute\_back$ is in $O(n^2)$ after Algorithm $\mathcal{DFS}$ terminates. □

6.2 Space Complexity

From Algorithms 1, 2, 3, and 4, we can trivially deduce the following theorem.

Theorem 10 *The memory requirement of Algorithm $\mathcal{UNNS} \circ \mathcal{DFS}$ is $O(n \times \log(\Delta) + \log(n))$ bits per processor where n is the number of processors and Δ is an upper bound on the degree of processors.*

7 Conclusion

We have presented a silent, distributed, and self-stabilizing algorithm which detects cut-nodes and bridges in arbitrary rooted networks. This algorithm must be composed with Algorithm $\mathcal{DFS}$ from [4]. After Algorithm $\mathcal{DFS}$ terminates, our algorithm needs only $O(H)$ rounds and $O(n^2)$ moves to reach a terminal configuration. This time complexity is equivalent to the best already proposed solutions. We have shown that the composite Algorithm $\mathcal{UNNS} \circ \mathcal{DFS}$ also works with an unfair daemon. Moreover, the memory requirement of Algorithm $\mathcal{UNNS} \circ \mathcal{DFS}$ is $O(n \times \log(\Delta) + \log(n))$ bits per processor. Until now, this is the protocol with the lowest memory requirement solving this problem.

References

[1] M Ahuja and Y Zhu. An efficient distributed algorithm for finding articulation points, bridges and biconnected components in asynchronous networks. In *9th Conference on Foundations of Software Technology and Theorical Computer Science, Banglore, India*, pages 99–108. LNCS 405, 1989.

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

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

[4] Z Collin and S Dolev. Self-stabilizing depth-first search. *Information Processing Letters*, 49(6):297–301, 1994.

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

[6] S Dolev, A Israeli, and S Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7:3–16, 1993.

[7] 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.

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

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

[10] J Parks, N Tokura, T Masuzawa, and K Hagihara. Efficient distributed algorithms solving problems about the connectivity of network. *Systems and Computers in Japan*, 22:1–16, 1991.

[11] K Paton. An algorithm for blocks and cutnodes of a graph. *Communications of the ACM*, 37:468–475, 1971.

[12] A P Spraque and K H Kulkarni. Optimal parallel algorithms for finding cuter vertices and bridges of internal graphs. *Information Processing Letters*, 42:229–234, 1992.

[13] Robert E Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Computing*, 1:No 2, june 1972.

[14] G Tel. *Introduction to distributed algorithms*. Cambridge University Press, 1994.