Parallel Processing Letters

A SILENT SELF-STABILIZING ALGORITHM FOR FINDING CUT-NODES AND BRIDGES

STÉPHANE DEVISMES
LaRIA, CNRS FRE 2733, Université de Picardie Jules Verne
80000 Amiens, France

Received (June 8, 2004)
Revised (December 13, 2004)
Communicated by (J. Beauquier)

ABSTRACT

In this paper, we present a self-stabilizing algorithm for finding cut-nodes and bridges in arbitrary rooted networks with a low memory requirement ($O(\log(n))$ bits per processor where n is the number of processors). Our algorithm is silent and must be composed with a silent self-stabilizing algorithm computing a Depth-First Search (DFS) Spanning Tree of the network. So, in the paper, we will prove that the composition of our algorithm with any silent self-stabilizing DFS algorithm is *self-stabilizing*. Finally, we will show that our algorithm needs $O(n^2)$ moves to reach a terminal configuration once the DFS spanning tree is computed. Note that 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* [1] is the most general technique to design a system to tolerate arbitrary transient faults. A self-stabilizing system, regarless of the initial states of the processors and messages initialy in the links, is guaranteed to converge to the intented behavior in finite time. 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 [2] and Tarjan [3], the latter has a linear time complexity. This problem has also been investigated in the context of parallel and distributed computing [4,5,6]. In self-stabilizing systems, Chaudhuri and Karaata present silent algorithms in [7,8,9,10] for finding cut-nodes and bridges. All these solutions work with an underlying spanning tree construction algorithm. The solutions proposed in [9,10] use a DFS spanning tree of the network while a BFS[a] spanning tree is required in [7,8]. Algorithms from [9,10] offer the best time complexity: they need $O(n^2)$ moves to stabilize once the DFS spanning tree is computed (instead of $O(n^2 \times m)$ moves for [7,8]). 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 locally maintain a set 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 a silent self-stabilizing algorithm computing a DFS spanning tree of the network. Until now, our protocol is the best algorithm solving this problem in term of memory requirement, i.e., $O(\log(n))$ bits per processor (without taking account of variables used for the spanning tree computing). Moreover, once the DFS spanning tree is computed, 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.

1.3. Outline of the paper

In the next section (Section 2), we describe the distributed system and the model in which our protocol is written. We present and prove our solution in Sections 3 and 4. In Section 5, we discuss about some complexity results. Finally, we make concluding remarks in Section 6.

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 are stored in the set $Neig_p$[b]. We assume that $Neig_p$ is a constant ($Neig_p$ is shown as an input from the system).

2.2. Computational Model

In the computation model that we use each processor executes the same program except

[a] Breath-First Search.

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

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 the system $\mathcal{S}$ is denoted as $\mathcal{E}$.

In a step of computation, first, all processors check the guards of their actions. Then, some *enabled* processors are chosen by a *daemon*. Finally, the "elected" processors execute one or more of theirs *enabled* actions. There exists several kinds of *daemon*. Here, we use a *central daemon*, i.e., during a computation step, if one or more processors are enabled, the daemon chooses one of these enabled processors to execute an action. Furthermore, we assume that the daemon is *unfair*, i.e., it can forever prevent a processor to execute an action except if it is the only enabled processor. The unfair daemon is the weakest scheduling assumption.

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

Definition 5 (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 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 (loopless) path 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 $T(r)$. $Children(p)$ 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 $\forall\, j$, 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 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 $T(r)$. We call *tree edges*, the edges of E' and *non-tree edges*, the edges of $E \setminus E'$. A *non-tree neighbor* of p is a node linked to p by a

non-tree edge. Finally, $\forall x \in V$, x is a *non-tree neighbor of a subtree* $T(p)$ if and only if $\exists y \in T(p)$ such that $(x, y) \in E \setminus E'$ (i.e., (x, y) is a non-tree edge of Tr)). For instance, in figure 1, Node 1 is a non-tree neighbor of $T(8)$.

Definition 7 (DFS Spanning Tree[c]) *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 trees.

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.

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.*

3. Algorithm

In this section, we present a silent self-stabilizing algorithm called Algorithm $\mathcal{UNNS}$[d]. Algorithms 1 and 2 formally describe Algorithm $\mathcal{UNNS}$. Our approach (Subsection 3.1) is mainly based on two results of Tarjan. These results concern the DFS spannning trees and their properties. Algorithm $\mathcal{UNNS}$ is informally describe in Subsection 3.3. This algorithm assumes that a DFS spanning tree of G (rooted at *r*) is available. We note $T(r)$=(V, E') this tree[e]. $T(r)$ can be obtained by applying any silent self-stabilizing DFS algorithm. Thus, in Subsection 3.2, we also discuss about the different silent self-stabilizing DFS algorithms of the literature. Finally, Algorithm $\mathcal{UNNS}$ runs concurrently with any silent self-stabilizing DFS algorithm following the *collateral composition* rules (as defined in [13]). So, in Subsection 3.4, we also recall the definition of this composition. In addition, we propose a general scheme for proving the correctness of a such composite algorithm (Theorem 3).

3.1. Approach

To implement our algorithm, we use two theorems established by Tarjan in [3]:

Theorem 1 *r (root of G) is a cut-node if and only if* $|Children(r)| \geq 2$.

Theorem 2 $\forall p \in V \setminus \{r\}$, p *is a cut-node if and only if there exists a node* $q \in Children(p)$ *for which no node in T(q) is linked by a non-tree edge to an ancestor of* p *in T(r).*

[c] This definition holds for undirected graphs only.
[d] Uppermost Non-tree Neighbor of each Subtree.
[e] The notations related to $T(r)$ introduced in Subsection 2.4 apply in this case as well.

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.

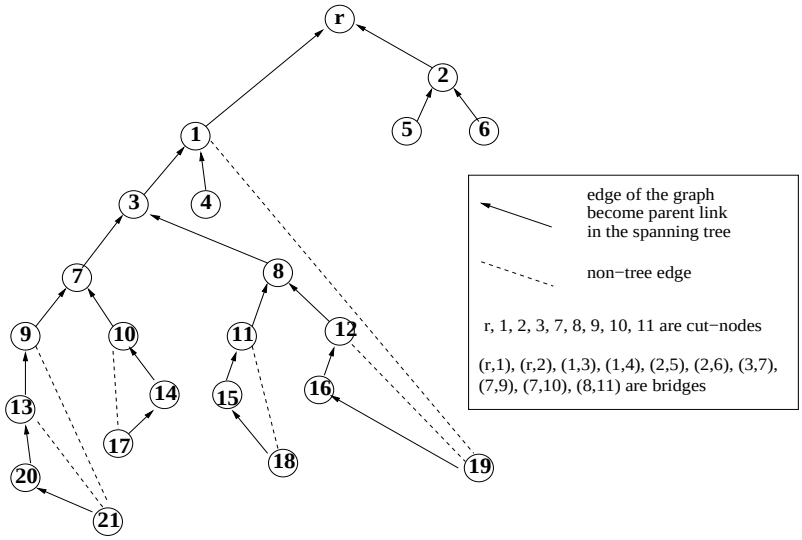

Fig. 1 DFS spanning tree in a connected undirected graph.

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

Remark 1 *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).*

From Theorem 2 and Remark 1, we know that the notion of non-tree neighbor is fondamental to determine cut-nodes and then bridges. So, in order to implement these ideas, we now introduce a new value $u(p)$:

$$\forall p \in V \setminus \{r\},\ u(p) = \min_{x \in T(p)}(\{\ h(y) :: (x,y) \in E - E'\} \cup \{h(x)\}),$$
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 $T(p)$. For example, in Figure 2, $T(4)$ has two non-tree neighbors: 4 and 6. Indeed, Node 4 is a non-tree neighbor of $T(4)$ because (4,6) is a non-tree edge of $T(r)$ and Node 6 $\in T(4)$. Conversely, Node 6 is a non-tree neighbor of $T(4)$ because (4,6) is a non-tree edge of $T(r)$ and Node 4 $\in T(4)$. Moreover, 4, 5, and 6 are the nodes of $T(4)$. Thus, $u(4) = \min(\{\ h(4), h(6)\} \cup \{\ h(4), h(5), h(6)\} = \min(\{3, 5, 3, 4, 6\}) = 3$.

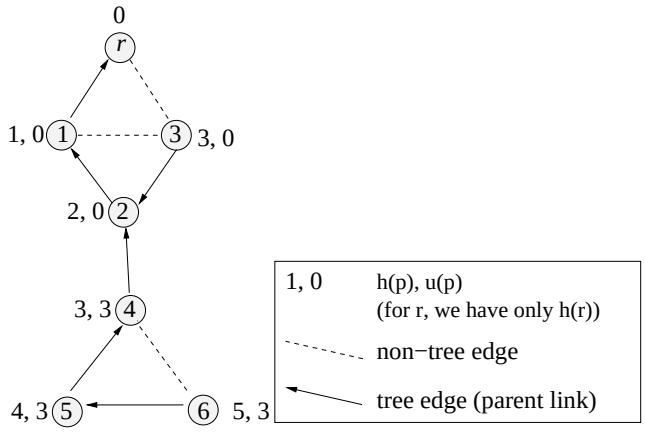

Fig. 2 Example on an arbitrary network.

3.1.1. Detection of Cut-nodes and Bridges

We now show the formal relation between the Tarjan's approach and $u(p)$. Roughly speaking, if $\forall p \in V$, we know $h(p)$, $u(p)$, $Parent(p)$ (for $p \neq r$), and $Children(p)$ 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) $\wedge$ ($|Children(p)| \geq 2$);*
2. *($p \neq$ r) $\wedge$ ($\exists q \in Children(p) :: u(q) \geq h(p)$).*

Proof. First, from Theorem 1, 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 2, 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 a DFS spanning tree of G, 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. □

3.1.2. Example

We now illustrated Propositions 1 and 2 with these following examples.

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.

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.

3.2. Algorithm $\mathcal{DFS}$

Obviously, to apply these concepts, we first need to computes a DFS spanning tree of the network. Several silent self-stabilizing DFS algorithms have been proposed in the literature, e.g., [14,15,16,17]. In particular, [17] have been proved assuming an unfair daemon. These Algorithms (e.g, [14,15,16,17]) have been written in different computation model but they can be easily adapted to our model because our computation model is simpler (As example, the adaptation of [15] is provided in the appendix). Moreover, using the scheme of proofs of [17], it is easy to prove that any adaptation of these algorithms in our computation model runs assuming an unfair daemon.

All these papers (e.g., [14,15,16,17]) are based on the same idea: each processor computes and stores a representation (using link index or ids) of the path of the DFS spanning tree leading to the root. Once the algorithm is stabilized, using this path, it is easy for all processor p to locally deduce several useful parameters[f]: $Parent(p)$ (only if $p \neq r$), $Children(p)$, and $h(p)$ without using any additionnal variable (i.e., without increasing the memory requirement of the DFS algorithm).

Thus, in the rest of the paper, we assume the existence of a silent DFS algorithm running under an unfair daemon and called Algorithm $\mathcal{DFS}$. Moreover, we assume that Algorithm $\mathcal{DFS}$ provides three macros Par_p (only for $p \neq r$), C_p, and $Height_p$. These macros computes $Parent(p)$, $Children(p)$, and $h(p)$ respectively. Obviously, these macros guarantee the expected result, i.e., $Par_p = Parent(p)$, $C_p = Children(p)$, and $Height_p = h(p)$, only when Algorithm $\mathcal{DFS}$ is stabilized. During the stabilization, we only assume that $Par_p \in Neig_p \cup \{\perp\}$, $C_p \subseteq Neig_p$, and $Height_p \in \mathbb{N}$. $Par_p = \perp$ means that p cannot currently distinguish any neighbor as its parent in the DFS spanning tree.

[f] See Algorithms 3 and 4 of the appendix

3.3. Algorithm $\mathcal{UNNS}$

Roughly speaking, Algorithm $\mathcal{UNNS}$ is a self-stabilizing distributed protocol for computing two predicates: Predicates $CutNote_p$ ($\forall p \in V$) and $ParentLink\_is\_a\_Bridges_p$ ($\forall p \in V \setminus \{r\}$). For the root, using Theorem 1, it is easy to deduce if it is a cut-node since Algorithm $\mathcal{DFS}$ is stabilized. For the other processors ($p \neq r$), Algorithm $\mathcal{UNNS}$ only consists to computes $u(p)$ in $Back_p$. With $u(p)$ and using Propositions 1 and 2, we can easily deduce (1) all the non-root cut-nodes of G and (2) all bridges of G. To that goal, in our algorithm, we use Macros Par_p (for $p \neq r$), C_p, and $Height_p$ as inputs from Algorithm $\mathcal{DFS}$ to know the parent of each node in $T(r)$, as well as their children, and $h(p)$ respectively. Macro $Nontree\_height_p$ collects the currently estimated height of each non-tree neighbor of p. Macro $Children\_back_p$ collects the currently estimated value of $u(q)$ for each children q of p in $T(r)$. So, if Macros $Nontree\_height_p$, $Children\_back_p$, and $Height_p$ contain the expected values, then, by definition, $u(p) = \min(Children\_back_p \cup Nontree\_height_p \cup \{height_p\})$. Hence, Algorithm $\mathcal{UNNS}$ has only one action called $Compute\_back_p$: if $Back_p \neq \min(Children\_back_p \cup Nontree\_height_p \cup \{height_p\})$, then p executes $Back_p := \min(Children\_back_p \cup Nontree\_height_p \cup \{height_p\})$. Finally, once the system is stabilized, according to Propositions 1 and 2, the predicates $CutNote_p$ ($\forall p \in V$) and $ParentLink\_is\_a\_Bridges_p$ ($\forall p \in V \setminus \{r\}$) allow to determine cut-nodes and bridges of the network, respectively.

As a consequence, in Section 4, we will simply prove that, for all possible executions, Algorithm $\mathcal{UNNS}$ reaches a terminal configuration and satisfies the following predicate in this configuration:

$$PRE_{UNNS} \equiv (\forall p \in V \setminus \{r\}, Back_p = u(p)).$$

Algorithm 1 Algorithm $\mathcal{UNNS}$ for $p = r$

Input:
C_p, $Height_p$: macros from $\mathcal{DFS}$;
$Neig_p$: set of neighbors;
Predicate:
$CutNode_p \equiv (|C_p| \geq 2)$

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

Input:
Par_p, C_p, $Height_p$: macros from $\mathcal{DFS}$;
$Neig_p$: set of neighbors;
Variable: $Back_p \in \mathbb{N}$;
Macro:
$Children\_back_p = \{x \in \mathbb{N} :: (\exists q \in C_p :: Back_q = x)\}$;
$Nontree\_height_p = \{x \in \mathbb{N} :: (\exists q \in Neig_p :: q \notin C_p \wedge Par_p \neq q \wedge Height_q = x)\}$;
Predicates:
$Update\_back(p) \equiv (Back_p \neq \min(Children\_back_p \cup Nontree\_height_p \cup \{Height_p\}))$
$CutNode_p \equiv (\exists q \in C_p :: Back_q \geq Height_p)$
$ParentLink\_is\_a\_Bridge_p \equiv (Back_p = Height_p)$
Actions:
$Compute\_back(p)$ :: $Update\_back(p) \rightarrow Back_p := \min(Children\_back_p \cup Nontree\_height_p \cup \{Height_p\})$;

3.4. Protocol Composition

Algorithm $\mathcal{UNNS}$ and Algorithm $\mathcal{DFS}$ run concurrently. For Algorithm $\mathcal{UNNS}$, Algorithm $\mathcal{DFS}$ is shown as a black box providing several outputs: Macros Par_p, C_p, and $Height_p$. Algorithm $\mathcal{UNNS}$ access to these macros in read only. So, Algorithm $\mathcal{UNNS}$ and Algorithm $\mathcal{DFS}$ are composed following *collateral composition* which defined in [13] as follows:

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.*

From now on, we note Algorithm $\mathcal{UNNS} \circ \mathcal{DFS}$ the collateral composition of Algorithm $\mathcal{UNNS}$ and Algorithm $\mathcal{DFS}$. The next definitions and theorem give the general scheme for the proof of self-stabilization of Algorithm $\mathcal{UNNS} \circ \mathcal{DFS}$.

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 3 *$S_2 \circ S_1$ 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.*

4. 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 first recall that the daemon is assumed to be unfair. Moreover, we assume that Algorithm $\mathcal{DFS}$ is silent and self-stabilizing, tolerates an unfair daemon, and outputs three macros: Par_p (for $p \neq r$ only), $Height_p$, and C_p.

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

Proof. We have assumed that Algorithm $\mathcal{DFS}$ is silent and runs assuming an unfair daemon. So, every execution of Algorithm $\mathcal{DFS}$ has a finite number of moves. Hence, we have just to prove that, between two configurations 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 (for $p \neq r$), C_p, $Height_p$ are set. Let $G_c = (V, E_c)$ be the partial directed graph of G where $E_c = \{(p,q) \in E :: p \neq r \wedge Par_p = q\}$ for an arbitrary configuration of Algorithm $\mathcal{DFS}$. Thus, G_c is the partial directed graph of G computed by Algorithm $\mathcal{DFS}$. We focus on G_c because, from Predicate $Update\_back$, we know that the updating of $Back_p$ for a processor p can only *cause* the execution of $Compute\_back$ of the processor pointed out by the macro Par_p, if it exists (remember that Par_p may be equal to $\perp$).
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 out 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. First, as $\forall p \in V_i$, Par_p pointed out, at most, one processor and CC_i is connected, we can remark that:

   (a) CC_i contains one cycle only. We note C this cycle and c its number of processors.

   (b) $\forall p \in V_i$, p is not a sink, i.e., a processor with no outgoing edge. In particular, that means that $p \neq r \wedge Par_p \neq \perp$.

   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 out 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 out 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, by 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 a DFS spanning tree $T(r)$ of G, the macro C_p equals $Children(p)$, and the macro $Height_p$ returns $h(p)$. 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, $\forall\ p \in V \setminus \{r\}$, $Back_p = u(p)$. 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, $Nontree\_height_f = \bigcup_{y \in Neig_f \setminus \{Par_f\}} \{h(y)\}$, $Children(f) = \emptyset$, 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, $Children\_back_q = \bigcup_{z \in Children(q)} \{u(z)\} = \bigcup_{z \in Children(q)} \{\min_{x \in T(z)}(\{h(y) :: (x, y) \in E - E'\} \cup \{h(x)\})\}$. $Nontree\_height_q = (\bigcup_{y \in Neig_q \setminus Children(q) \setminus \{Par_q\}} \{h(y)\}) = \{h(y) :: (q, y) \in E - E'\}$. Thus, $Back_q = \min(\bigcup_{z \in Children(q)} \{\min_{x \in T(z)}(\{h(y) :: (x, y) \in E - E'\} \cup \{h(x)\})\} \cup \{h(y) :: (q, y) \in E - E'\} \cup \{h(q)\}) = \min(\bigcup_{z \in Children(q)} (\bigcup_{x \in T(z)}(\{h(y) :: (x, y) \in E - E'\} \cup \{h(x)\})) \cup \{h(y) :: (q, y) \in E - E'\} \cup \{h(q)\})$. Now, $T(q) = (\bigcup_{z \in Children(q)} \{T(z)\}) \cup \{q\}$. 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 4 *Algorithm $\mathcal{UNNS} \circ \mathcal{DFS}$ stabilizes to PRE_{UNNS} under the unfair daemon.*

Proof. From the following four observations and Theorem 3, the result holds.

1. By assumption, 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).

□

Finally, from Theorem 4, Propositions 1 and 2, we can claim the following theorem.

Theorem 5 *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 $CutNode_p$.*

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 p satisfies $ParentLink\_is\_a\_Bridge_p$.*

5. 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 a DFS spanning tree of G, $T(r)$. We also presented the space complexity of our solution.

5.1. Time Complexity

Theorem 6 *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, a DFS tree of G, $T(r)$, has been computed: C_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 on. 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 7 *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 confi guration.
- A change in the state of a neighbor.

We call an *initial* move: a move caused by an initial confi guration. As we assume Algorithm $\mathcal{DFS}$ is terminated, we can consider that, $\forall\ p$, the values returned by the macros Par_p (for $p \neq r$), C_p, and $Height_p$ are also set. Hence, there exists a DFS spanning tree of G 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. □

5.2. Space Complexity

Algorithm 1 contains no variable. Algorithm 2 contains on variable only: $Back$. $\forall p \in V \setminus \{r\}$, $Back_p$ is used to store the height of p or the height of one of its ancestor. So, the value of $Back_p$ can be bounded by n and the following theorem is obvious.

Theorem 8 *The memory requirement of Algorithm $\mathcal{UNNS}$ is $O(\log(n))$ bits per processor where n is the number of processors.*

6. 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 a silent self-stabilizing algorithm computing a DFS spanning tree of the network like [16,17]. Once the DFS spanning tree is computed, our algorithm needs only $O(H)$ rounds and $O(n^2)$ moves to reach a terminal confi guration. This time complexity is equivalent to the best already proposed solutions. We have shown that our solution works under an unfair

daemon, i.e., the weakest scheduling assumption. Moreover, the memory requirement of our algorithm is $O(\log(n))$ bits per processor (without taking account of variables used for the spanning tree computing). Until now, this is the protocol with the lowest memory requirement solving this problem.

References

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

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

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

[4] 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.

[5] 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.

[6] 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.

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

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

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

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

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

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

[14] T Herman. *Adaptivity through distributed convergence*. PhD thesis, University of Texas at Austin, Departement of Computer Science, 1991.

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

[16] Y Afek and A Bremler. Self-stabilizing unidirectional network algorithms by power supply. *Chicago Journal of Theoretical Computer Science*, 1998:Article 3, 1998.

[17] B Ducourthial and S Tixeuil. Self-stabilization with r-operators. *Distributed Computing*, pages 14(3):147–162, July 2001.

Appendix A: Adaptation for the state model of the DFS algorithm of Collin and Dolev

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

Input: $Neig_p$: set of neighbors (locally ordered);

Constant: $Path_p$=$\perp$;

Macro:
$Height_p = |Path_p| - 1$;
$C_p = \{q \in Neig_p :: (Path_p \oplus \alpha_p(q)) = Path_q\}$;

Algorithm 4 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:
$Read\_path_p = \bigcup_{q \in Neig_p} \{\text{right}_N(Path_q \oplus \alpha_q(p))\}$;
$Height_p = |Path_p| - 1$;
$Par_p = q$ **if** $(\exists!\ q \in Neig_p :: (Path_q \oplus \alpha_q(p)) = Path_p)$, $\perp$ **otherwise**;
$C_p = \{q \in Neig_p :: (Path_p \oplus \alpha_p(q)) = Path_q\}$;

Predicates:

$Update\_path(p) \equiv (Path_p \neq \min_{\prec_{lex}}(Read\_path_p))$

Action:

$Compute\_path(p)$ :: $Update\_path(p) \rightarrow Path_p := \min_{\prec_{lex}}(Read\_path_p)$;

First, in [15], authors assume that $\forall p \in V$, $\forall q \in Neig_p$, p knows $\alpha_q(p)$, i.e., the index of Edge (p, q) in $Neig_q$ (for details see [15]). Then, 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 computes its *path* field according to the *path* fields of its neighbors. From the path $Path_q$, read by p from the neighbor q, p derives a path by concatening $Path_q$ with $\alpha_q(p)$ (noted $Path_q \oplus \alpha_q(p)$). Then, p chooses its path to be the minimal path (according to the lexicographical order $\prec_{lex}$) 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 elementary path that is possible in G. Thus, the notation right$_k(w)$ refers to the sequence of the k least significant items of w.