Self-Stabilizing Labeling and Ranking in Ordered Trees*

Ajoy K. Datta[1], Stéphane Devismes[2], Lawrence L. Larmore[1], and Yvan Rivierre[2]

[1] School of Computer Science, University of Nevada Las Vegas, USA,
firstname.lastname@unlv.edu,
WWW home page: http://www.egr.unlv.edu/~lastname
[2] VERIMAG UMR 5104, Université Joseph Fourier, France,
firstname.lastname@imag.fr,
WWW home page: http://www-verimag.imag.fr/~lastname

Abstract. We propose two self-stabilizing algorithms for tree networks. The first one computes a special label, called *guide pair* of each process P in $O(h)$ rounds (h being the height of the tree) using $O(\delta_P \log n)$ space per process P, where δ_P is the degree of P and n the number of processes in the network. Guide pairs have numerous applications, including ordered traversal or navigation of the processes in the tree. Our second self-stabilizing algorithm, which uses the guide pairs computed by the first algorithm, solves the *ranking problem* in $O(n)$ rounds and has space complexity $O(b+\delta_P \log n)$ in each process P, where b is the number of bits needed to store a value. The first algorithm orders the tree processes according to their topological positions. The second algorithm orders (ranks) the processes according to the values stored in them.

Keywords: Self-stabilization, tree networks, tree labeling, ranking problem.

1 Introduction

Self-stabilization [4,5] is a versatile property, enabling an algorithm to withstand transient faults in a distributed system. A distributed algorithm is self-stabilizing if, after transient faults hit the system and place it in some arbitrary global state, the system recovers without external intervention in finite time.

An *ordered tree* $\mathcal{T}$ is a rooted tree, together with an order (called a left-to-right order) on the children of every node. In this paper, we give two self-stabilizing distributed algorithms for ordered trees. None of the two algorithms assumes knowledge of the size of the network n, or of a known upper bound of n, although, as it is usual in the literature, we assume that each process can store an integer in the range $1..n$, using $O(\log n)$ space. We choose the ordered tree topologies because results in such topologies can be easily extended to arbitrary rooted networks by composing our solutions with any existing self-stabilizing spanning tree construction algorithm (see [5] for the literature). However, the meaning of "traversing" or "ranking" processes in a general network is not clear.

* This work has been partially supported by the ANR project *ARESA*2.

Our first algorithm, GUIDE, computes a *guide pair* for each process P, which we write as $P.guide = (P.pre\_ind, P.post\_ind)$, where $P.pre\_ind$ and $P.post\_ind$ are the rank of P in the *preorder* and *reverse postorder* traversal, respectively, of the ordered tree. Figure 1 shows an example of ordered tree labeled with guide pairs. The guide pairs provide a labeling scheme that can be used for various applications [7]. In this work, we use these labels to navigate in the tree $\mathcal{T}$. We can define a partial ordering on the guide pairs as follows: We say $(i, j) \leq (k, \ell)$ if $i \leq k$ and $j \leq \ell$. Then, A process Q is a member of the subtree $\mathcal{T}_P$ rooted at P if and only if $P.guide \leq Q.guide$. The guide pairs can be used to implement routing between any two processes of the tree. If the two nodes satisfy the above partial ordering, then the routing path simply follows the list of ancestors/descendants. Otherwise, the routing must be established via the nearest common ancestor.

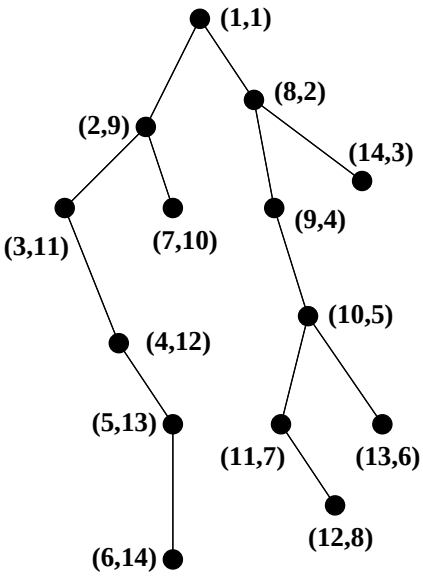

Fig. 1. Guide pairs.

Our second algorithm, RANK, uses GUIDE, hence shows another application of guide pairs. The input of our second algorithm is a value $P.weight$, of some ordered type, for each process P. RANK computes the *rank* of each process, which is defined to be the index of that process if all processes were sorted by their weights.

1.1 Contributions

GUIDE has time complexity $O(h)$ rounds, where h is the height of $\mathcal{T}$. The time complexity of RANK is $O(n)$ rounds. The space complexity of GUIDE in each process P is $O(\delta_P \log n)$, where δ_P is the degree of P. RANK, which uses GUIDE as a subroutine, has space complexity $O(b + \delta_P \log n)$ in each process P, where b is the number of bits needed to store a value. GUIDE and RANK are self-stabilizing. GUIDE is *silent*, that is, it eventually reaches a terminal configuration where all actions of all processes are disabled. RANK correctly computes the rank of every process within $O(n)$ rounds. Unless the weights change, the ranking do not change once the system stabilizes. However, the algorithm repeatedly computes them to detect possible change of weights. If the weights do not change, the repeated computation of RANK will be transparent to the application that uses the output of RANK.

1.2 Related Work

The notion of guide pairs appeared first in [7], but that solution is not self-stabilizing. To the best of our knowledge, there exist no self-stabilizing algorithms for computing the guide pairs.

The only self-stabilizing solution to the ranking problem was given in [2]. This algorithm works in rooted trees. Like ours, that algorithm is not silent. Moreover, it assumes that each process has a unique identifier in the range $1..n$. The algorithm stabilizes in $\Omega(n^2)$ rounds using $O(\log n)$ space per process. The ranking problem is related to the *sorting problem*. There exist numerous self-stabilizing solutions to sorting in a tree, *e.g.*, [9,8,1]. However, all those previous problems are quite different than ours.

1.3 Roadmap

In the next section, we present the model we use throughout this paper. In Section 3, we present our self-stabilizing silent algorithm for computing guide pairs. In Section 4, we present our self-stabilizing algorithm for the ranking problem, which uses the guide pairs. Because of space limitations, the proofs have been omitted. See the technical report online (`http://www-verimag.imag.fr/~devismes/WWW/rapports/trRank.pdf`).

2 Preliminaries

Let $G = (V, E)$ be an undirected graph, where V is a set of nodes and E is a set of undirected edges linking nodes. Two nodes $P, Q \in V$ are said to be neighbors if $\{P, Q\} \in E$. The set of P's neighbors is denoted by $N(P)$. The degree of P *i.e.*, $|N(P)|$, is denoted by δ_P. $G = (V, E)$ is a *tree* if it is connected and acyclic. A tree $\mathcal{T}$ can be *rooted* at some node, meaning that one of its nodes $Root$ is distinguished as the *root* (all other nodes are anonymous). In a rooted tree $\mathcal{T}$, we denote by $P.par$, the parent of node P in $\mathcal{T}$: If $P = Root$, then $P.par = P$; otherwise $P.par = Q$, where Q is the neighbor of P that is the closest from the root (in this case, P is said to be a *child* of Q in $\mathcal{T}$). Let $Chldrn(P) = \{Q \in N(P) : Q.par = P\}$, the *children* of P in the tree $\mathcal{T}$. An *ordered tree* is a rooted tree $\mathcal{T}$, together with an (local) order (called a left-to-right order) on the children of every node. We denote by $\prec_P$ the local order relation among the children of node P. Let $P_1, P_2, \ldots P_m$ be the children of the root of $\mathcal{T}$ in the left-to-right order. We denote by $\mathcal{T}_i$ be the subtree rooted at any P_i. Finally, we denote by $Q \in \mathcal{T}_i$ the fact that node Q is a node of $\mathcal{T}_i$.

We model our network topology as an ordered tree $\mathcal{T} = (V, E)$, where V is a set of n nodes representing processes and E is a set of edges, each representing the ability of two processes to communicate directly. (We will use the terms "node" and "process" interchangeably.) We denote by $h(P)$ the height of process P in $\mathcal{T}$, *i.e.*, its distance to the root. We denote by h the height of $\mathcal{T}$, *i.e.*, $\max_{P \in V} h(P)$.

2.1 Computational Model

We consider the locally shared memory model, introduced by Dijkstra [4]. In this model, communications are carried out by locally shared variables. Each process has the finite set of shared variables (henceforth, referred to as variables) whose domains are finite. A process P can read its own variables and that of its neighbors, but can write only to its own variables. We assume that every process P can read the local names of its neighbors, so that if $Q \in N(P)$, P can tell, for example, whether $Q.par = P$. Each process writes its variables according to its (local) *program*. A *distributed algorithm* is a collection of n *programs*, each one operating on a single process. The *program* of each process is a finite set of actions $\langle label \rangle :: \langle guard \rangle \mapsto \langle statement \rangle$. *Labels* are only used to identify actions in the discussion. The *guard* of an action in the program of a process 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 it is *enabled*, *i.e.,* its guard evaluates to $true$. A process is said to be *enabled* if at least one of its actions is enabled.

Let $\mathcal{A}$ be a distributed algorithm operating of a network of topology G. The values of $\mathcal{A}$'s variables at some process P define $\mathcal{A}$'s *(local) state* of P in G. A configuration of $\mathcal{A}$ in G is an instance of $\mathcal{A}$'s states of all processes in G. In the following, if there is no ambiguity, configurations of $\mathcal{A}$ in G will be simply denoted by *configurations*.

Let $\mapsto$ be the binary relation over configurations of $\mathcal{A}$ in G such that $\gamma \mapsto \gamma'$ if and only if it is possible for the network of topology G to change from configuration γ to configuration γ' in one step of $\mathcal{A}$. An *execution* of $\mathcal{A}$ is a maximal sequence of configurations $\varrho = \gamma_0\gamma_1 \ldots \gamma_i \ldots$ such that $\gamma_{i-1} \mapsto \gamma_i$ for all $i > 0$. The term "maximal" means that the execution is either infinite, or ends at a *terminal* configuration in which no action of any process is enabled. Each step $\gamma_i \mapsto \gamma_{i+1}$ consists of one or more *enabled* processes executing an action. The evaluations of all guards and executions of all statements of those actions are presumed to take place in one atomic step; this model is called *composite atomicity* [5].

We assume that each step from a configuration to another is driven by a *scheduler*, also called a *daemon*. If one or more processes are enabled, the scheduler selects at least one of these enabled processes to execute an action. We assume that the scheduler is *weakly fair*, meaning that, every continuously enabled process P is selected by the scheduler within finite time.

We say that a process P is *neutralized* in the step $\gamma_i \mapsto \gamma_{i+1}$ if P is enabled in γ_i and not enabled in γ_{i+1}, but does not execute any action between these two configurations. The neutralization of a process represents the following situation: at least one neighbor of P changes its state between γ_i and γ_{i+1}, and this change effectively makes the guard of all actions of P false.

We use the notion of *round*. The first *round* of an execution ϱ, noted ϱ', is the minimal prefix of ϱ in which every process that is enabled in the initial configuration either executes an action or becomes neutralized. Let ϱ'' be the suffix of ϱ starting from the last configuration of ϱ'. The second round of ϱ is the first round of ϱ'', the third round of ϱ is the second round of ϱ'', and so forth.

2.2 Self-stabilization and Silence

In the following, we define a *specification* as a set of executions. We said that an execution ϱ satisfies the specification SP if $\varrho \in SP$.

A distributed algorithm $\mathcal{A}$ is *self-stabilizing with respect to* the specification SP in a network of topology G if and only if there exists a set of configurations $\mathcal{C}$ such that:

1. Every execution of $\mathcal{A}$ in a network of topology G starting from a configuration in $\mathcal{C}$ satisfies SP (*closure*).
2. Every execution of $\mathcal{A}$ in a network of topology G eventually reaches a configuration in $\mathcal{C}$ (*convergence*).

All configurations of $\mathcal{C}$ are said to be legitimate, all other configurations are said to be illegitimate.

We say that an algorithm is *silent* [6] if each of its executions is finite. In other words, starting from an arbitrary configuration, the network will eventually reach a configuration where no process is enabled.

2.3 Composition

To simplify the design of our algorithms, we use a variant of the well-known *collateral composition* [10]. Roughly speaking, when we collaterally compose two algorithms $\mathcal{A}$ and $\mathcal{B}$, $\mathcal{A}$ and $\mathcal{B}$ run concurrently and $\mathcal{B}$ uses the outputs of $\mathcal{A}$ in its executions. In the variant we use, we modify the code of $\mathcal{B}$ so that a process executes an action of $\mathcal{B}$ only when it has no enabled action in $\mathcal{A}$.

Let $\mathcal{A}$ and $\mathcal{B}$ be two algorithms such that no variable written by $\mathcal{B}$ appears in $\mathcal{A}$. The *hierarchical collateral composition* [3] of $\mathcal{A}$ and $\mathcal{B}$, noted $\mathcal{B} \circ \mathcal{A}$, is the algorithm defined as follows:

1. $\mathcal{B} \circ \mathcal{A}$ contains all variables of $\mathcal{A}$ and $\mathcal{B}$.
2. $\mathcal{B} \circ \mathcal{A}$ contains all actions of $\mathcal{A}$.
3. For every action "$L_i :: G_i \mapsto S_i$" of $\mathcal{B}$, $\mathcal{B} \circ \mathcal{A}$ contains the action "$L_i :: \neg D \wedge G_i \mapsto S_i$" where D is the disjunction of all guards of actions in $\mathcal{A}$.

The following sufficient condition is given in [3] to show the correctness of the composite algorithm:

Theorem 1. *The composite algorithm $\mathcal{B} \circ \mathcal{A}$ self-stabilizes to specification SP in a network of topology G assuming a weakly fair scheduler if the following conditions hold: (i) in a network of topology G, Algorithm $\mathcal{A}$ is a silent algorithm under a weakly fair scheduler; (ii) in a network of topology G, Algorithm $\mathcal{B}$ stabilizes to SP under a weakly fair daemon, starting from any configuration where no action of $\mathcal{A}$ is enabled.*

3 Computing Guide Pairs

3.1 Guide Pairs

Given an ordered tree $\mathcal{T}$, the guide pair of a node P in $\mathcal{T}$ is the pair of integers i and j such that i and j are, respectively, the rank of P in the *preorder* and *reverse postorder* traversal of $\mathcal{T}$. Below, we define these notions. Recall that we denote by $P_1, P_2, \ldots P_m$ the children of the root of $\mathcal{T}$ in the left-to-right order, and we denote by $\mathcal{T}_i$ be the subtree rooted at any P_i. The *preorder traversal* of $\mathcal{T}$ is defined, recursively, as follows:

1. Visit the root of $\mathcal{T}$.
2. For each i from 1 to m in increasing order, visit the nodes of $\mathcal{T}_i$ in *preorder*.

Postorder traversal $\mathcal{T}$ is similarly defined:

1. For each i from 1 to m in increasing order, visit the nodes of $\mathcal{T}_i$ in *postorder*.
2. Visit the root of $\mathcal{T}$.

Preorder traversal is top-down, while postorder traversal is bottom-up. However, we can also traverse $\mathcal{T}$ in *reverse postorder*, which is top-down, as follows.

1. Visit the root of $\mathcal{T}$.
2. For i from m to 1 in decreasing order, visit the nodes of $\mathcal{T}_i$ in *reverse postorder*.

If a node P is the i^{th} node of $\mathcal{T}$ visited in a preorder traversal of $\mathcal{T}$, we say that the *preorder rank* of P is i. If a node P is the j^{th} node of $\mathcal{T}$ visited in a reverse postorder traversal of $\mathcal{T}$, we say that the *reverse postorder rank* of P is j. Write *pre_ind*(P) and *post_ind*(P) for the *preorder* rank and *reverse postorder* rank of P, respectively. We define the *guide pair* of P to be the ordered pair *guide*$(P) =$ (*pre_ind*(P), *post_ind*(P)). Figure 1 shows an ordered tree where each process is labeled with its guide pair.

If (i, j) and (k, ℓ) are guide pairs, we write $(i, j) \leq (k, \ell)$ if $i \leq k$ and $j \leq \ell$. Thus, the set of guide pairs is partially ordered by $\leq$.

Remark 1. [Property 2 in [7]] If P and Q are nodes of an ordered tree $\mathcal{T}$, then *guide*$(P) \leq$ *guide*(Q) if and only if P is an ancestor of Q.

3.2 Algorithm GUIDE

Algorithm GUIDE is a hierarchical collateral composition of two algorithms: GUIDE $=$ CGP $\circ$ COUNT, where both COUNT and CGP (for *Compute Guide Pairs*) use P*.par* as input in the program of every process P. Note that P*.par* either designates the actual parent link of P or is computed by a distributed spanning tree algorithm with which GUIDE must be composed using the hierarchical collateral composition.

Algorithm COUNT**.** COUNT acts as a bottom-up wave that computes the number of processes in each subtree. In COUNT, each process P has only one variable: P.*subcount*. Moreover, each process P can compute the following function: *Subcount*$(P) = 1 + \sum_{Q \in \textit{Chldrn}(P)} Q.\textit{subcount}$. Thus, the program of P consists of the following action:

$$\texttt{SetCnt} :: P.\textit{subcount} \neq \textit{Subcount}(P) \mapsto P.\textit{subcount} \leftarrow \textit{Subcount}(P)$$

Lemma 1. COUNT *is self-stabilizing and silent, converges within* $h+1$ *rounds from an arbitrary initial configuration to a legitimate configuration where* $P.\textit{subcount} = |\{Q \in \mathcal{T}_P\}|$ *for all processes* P*, and works under the weakly fair scheduler.*

Algorithm CGP**.** Using the values of *subcount* computed by COUNT, each process P evaluates in CGP for each of its children Q the number of processes before Q in the *preorder* and *reverse postorder* traversal of the tree $\mathcal{T}$, respectively (using Actions `SetChldPrePred` and `SetChldPostPred`, respectively). Then, reading these values from its parent, each process, except the root, can compute its guide pair (using Actions `SetPreInd` and `SetPostInd`). The guide pair of the root is $(1, 1)$ (see Actions `SetPreInd` and `SetPostInd` for the root).

Variables of CGP. In CGP, each process maintains several variables. First, the following array variable enables each non-root process to know its index in the local left-to-right order of its parent:

1. $P.\textit{chld}[i] \in N(P) \cup \{\perp\}$, for all $1 \leq i \leq \delta_P$. This array is maintained by Action `SetChld`. For all $1 \leq i \leq |\textit{Chldrn}(P)|$, $P.\textit{chld}[i]$ is set to the i^{th} child in P's local ordering of $N(P)$, while for all $|\textit{Chldrn}(P)| < i \leq \delta_P$, $P.\textit{chld}[i]$ is set to $\perp$.[3]

Then, CGP uses the following additional variables:

2. $P.\textit{pre\_ind}$, $P.\textit{post\_ind}$, integers. In stabilized state, they contain the *preorder* and *reverse postorder* ranks of P, respectively. Thus, we will write $P.\textit{guide} = (P.\textit{pre\_ind}, P.\textit{post\_ind})$, the guide pair of P.
3. $P.\textit{chld\_pre\_pred}[i]$, $P.\textit{chld\_post\_pred}[i]$, integer, defined for all $1 \leq i \leq |\delta_P|$:
   - For all $1 \leq i \leq |\textit{Chldrn}(P)|$, $P.\textit{chld\_pre\_pred}[i]$ is set to the number of predecessors of the i^{th} child of P (that is, $P.\textit{chld}[i]$) in the *preorder* traversal of $\mathcal{T}$; and $P.\textit{chld\_post\_pred}[i]$ is set to the number of predecessors of the i^{th} child of P in the *reverse postorder* traversal of $\mathcal{T}$.
   - For all $|\textit{Chldrn}(P)| < i \leq \delta_P$, $P.\textit{chld\_pre\_pred}[i]$ and $P.\textit{chld\_post\_pred}[i]$ are set to 0.

   Hence, each process P computes its guide pair to be

$$(P.\textit{par}.\textit{chld\_pre\_pred}[j] + 1, P.\textit{par}.\textit{chld\_post\_pred}[j] + 1)$$

   where P is the j^{th} child of its parent in left-to-right order.

[3] Actually, cells from index $|\textit{Chldrn}(P)| + 1$ to δ_P are useless. However, as the tree may be obtained by a spanning tree construction, we cannot know the number of children of P in advance, but this number is bounded by δ_P.

Functions of CGP. Based on the previous variables, each process P can compute the following functions:

- *my_order*(P). If P is not the root and there exists i, $1 \leq i \leq \delta_{P.par}$, such that $P.par.chld[i] = P$, then *my_order*(P) returns i. If the values of $P.par.chld$ did not stabilize, *my_order*(P) returns 1.
  Once the system has stabilized, *my_order*(P) returns the index of the non-root process P in the local left-to-right order of its parent.
- $Chld\_index(Q) = |\{Q' \in Chldrn(P) : Q' \prec_P Q\}| + 1$. It returns the index of the child Q of process P in the local left-to-right order of P.
- $Eval\_chld(i)$ returns the local name of the i^{th} child of P. That is, if $\exists Q \in Chldrn(P)$ such that $Chld\_index(Q) = i$, then $Eval\_chld(i)$ returns Q; otherwise, $Eval\_chld(i)$ returns $\perp$.
- $Eval\_chld\_pre\_pred(i)$. If $i = 1$, then $Eval\_chld\_pre\_pred(i)$ returns $P.pre\_ind$; else if $2 \leq i \leq |Chldrn(P)|$, then $Eval\_chld\_pre\_pred(i)$ returns $P.chld\_pre\_pred[i-1] + P.chld[i-1].subcount$; otherwise it returns 0.
  Once the system has stabilized, $Eval\_chld\_pre\_pred(i)$ returns the number of predecessors of the i^{th} child of P in the *preorder* traversal of $\mathcal{T}$.
- $Eval\_chld\_post\_pred(i)$. If $i = |Chldrn(P)|$, then $Eval\_chld\_post\_pred(i)$ returns $P.post\_ind$; else if $1 \leq i < |Chldrn(P)|$, then $Eval\_chld\_post\_pred(i)$ returns $P.chld\_post\_pred[i+1] + P.chld[i+1].subcount$; otherwise $Eval\_chld\_post\_pred(i)$ returns 0.
  Once the system has stabilized, $Eval\_chld\_post\_pred(i)$ returns the number of predecessors of the i^{th} child of P in the reverse postorder traversal of $\mathcal{T}$.

Actions of CGP. Actions of CGP are given below. To simplify the presentation, we assume priorities on actions, and list them below in the order from the highest to the lowest priority. If several actions are enabled simultaneously at a process, only the one of the highest priority can be executed. In other words, the actual guard of any action "$L :: G \mapsto S$" of process P is $\neg D \wedge G$, where D is the disjunction of the guards of all actions at P that appear before in the text.

For every process P:

`SetChld` :: $\exists i \in [1..\delta_P]$, $P.chld[i] \neq Eval\_chld(i)$ $\mapsto$ $\forall i \in [1..\delta_P]$, $P.chld[i] \leftarrow Eval\_chld(i)$

`SetChldPrePred` :: $\exists i \in [1..\delta_P]$, $P.chld\_pre\_pred[i] \neq Eval\_chld\_pre\_pred(i)$ $\mapsto$ $\forall i \in [1..\delta_P]$, $P.chld\_pre\_pred[i] \leftarrow Eval\_chld\_pre\_pred(i)$

`SetChldPostPred` :: $\exists i \in [1..\delta_P]$, $P.chld\_post\_pred[i] \neq Eval\_chld\_post\_pred(i)$ $\mapsto$ $\forall i \in [1..\delta_P]$, $P.chld\_post\_pred[i] \leftarrow Eval\_chld\_post\_pred(i)$

For the root process $Root$ only:

`SetPreInd` :: $Root.pre\_ind \neq 1 \mapsto Root.pre\_ind \leftarrow 1$

`SetPostInd` :: $Root.post\_ind \neq 1 \mapsto Root.post\_ind \leftarrow 1$

For every non-root process P only:

`SetPreInd` :: $P.pre\_ind \neq 1 + P.par.chld\_pre\_pred[my\_order(P)] \mapsto P.pre\_ind \leftarrow 1 + P.par.chld\_pre\_pred[my\_order(P)]$

`SetPostInd` :: $P.post\_ind \neq 1 + P.par.chld\_post\_pred[my\_order(P)] \mapsto P.post\_ind \leftarrow 1 + P.par.chld\_post\_pred[my\_order(P)]$

Overview of CGP. We now give an intuitive explanation of how CGP computes the values of P.*pre_ind* for all P. The values of P.*post_ind* are computed similarly.

Suppose that P is the i^{th} process visited in a *preorder* traversal of $\mathcal{T}$. Then i is the correct value of P.*pre_ind*. CGP works by computing the number of predecessors of P, *i.e.*, the number of processes visited before P is visited. Let us call that number *Num_Preorder_Preds*(P). It is the correct value of P.*pre_ind* $- 1$.

Num_Preorder_Preds$(Root) = 0$; otherwise, *Num_Preorder_Preds*(P) is computed by P.*par* and stored in the variable P.*par*.*chld_pre_pred*$[j]$, where P is the j^{th} child of P.*par* in left-to-right order. In order to compute these values for all its children, P.*par* must have computed its own value of *pre_ind*, as well as the sizes of all of its subtrees. If $j = 1$, then *Num_Preorder_Preds*(P) $=$ P.*par*.*pre_ind*, since P.*par* is the immediate predecessor of its leftmost child in the *preorder* visitation. Thus, P.*par*.*chld_pre_pred*$[1] \leftarrow P$.*par*.*pre_ind*. P.*par*.*chld_pre_pred*$[2]$ is obtained by adding the size of the leftmost subtree of P.*par* to P.*par*.*chld_pre_pred*$[1]$, since all members of that subtree are predecessors of the second child of P.*par*.

In general, the number of predecessors of P is equal to P.*par*.*pre_ind* plus the sum of the sizes of the leftmost $j - 1$ subtrees of P.*par*. The values of the array P.*par*.*chld_post_pred* are computed from right to left, similarly. P then executes:

$$P.\textit{pre\_ind} \leftarrow P.\textit{par}.\textit{chld\_pre\_pred}[j] + 1$$
$$P.\textit{post\_ind} \leftarrow P.\textit{par}.\textit{chld\_post\_pred}[j] + 1$$

Theorem 2. GUIDE *is self-stabilizing and silent, computes the guide pairs of all processes in $O(h)$ rounds from an arbitrary initial configuration, and works under the weakly fair scheduler.*

4 Rank Ordering

In this section, we give an algorithm, RANK, that uses guide pairs to solve the *ranking problem* on an ordered tree, $\mathcal{T}$. We are given a value P.*weight* for each process P in $\mathcal{T}$. (For convenience, we assume that the weights are integers.) The problem is to find the *rank* of each P. If $P_1, P_2, \ldots, P_n$ is the list of processes in $\mathcal{T}$ sorted by weight, then i is the rank of P_i. We allow ties to be broken arbitrarily, but deterministically.

Our algorithm RANK is a hierarchical collateral composition of two algorithms: RANK $=$ CRK $\circ$ GUIDE. RANK computes the rank of each process P in $\mathcal{T}$, and sets the variable P.*rank* to that value. RANK is self-stabilizing, and requires $O(n)$ rounds and $O(b + \delta_P \log n)$ space for each process P.

4.1 Overview of CRK

Flow of Packages. The key part of the algorithm CRK is the *flow of packages*. Each package is an ordered pair $x = (x.\textit{value}, x.\textit{guide})$, where x.*value* is its *value* and x.*guide* is its *guide pair*. We identify a package with its *guide pair*. Moreover, for every two packages, x and y, we have $x \geq y$ (resp. $x > y$) if and only if $x.\textit{value} \geq y.\textit{value}$ (resp. $x.\textit{value} > y.\textit{value}$).

Each package has a *home process* (the node from which the package is originally issued), although its location can be at any process in the chain between its home and the root. The guide pair of a package is the same as the guide pair of its home process, and its value is either the weight of its home process or the rank that CRK will assign to its home process.

Each process P initiates its flow of packages by creating an *up-package* whose value is P.*weight*. This up-package then moves to the root by successive copying. The flow of packages is organized so that packages with smaller weights reach the root before packages with larger weights, in a manner similar to the standard technique for maintaining min-heap order in a tree.

After the root copies an up-package from a child, it creates a *down-package* with the same home process as the up-package, but whose value is a number (a rank) in the range $1..n$. The root maintains a counter so that the first down-package it creates has value 1, the second value 2, and so forth. Each down-package then moves back to its home process by copying. When its home process copies a down-package, it assigns, or re-assigns, its rank to be the value of that package.

The purpose (in fact even the name) of the guide pair is now obvious. It is used to guide the down-package to its home process.

Since the root copies up-packages in weight order, it creates down-packages in that same order. The i^{th} down-package created by the root will carry rank i and will use the same guide pair as the i^{th} up-package copied by the root. Its home process will then be the process whose weight is the i^{th} smallest in $\mathcal{T}$.

When the root detects that it has created all down-packages, it initiates a broadcast wave which resets the variables of CRK (except the rank and weight variables) and starts a new epoch.

Redundant Packages. In our model of computation, if a variable of a process P is copied by a neighbor Q, it also remains at P. In the algorithm CRK, each process P can be home to at most one package, but we cannot avoid the existence of multiple copies of that package (up and/or down). We handle that problem by defining a package variable currently held by a process (not necessarily its home process, rather any process on the chain from its home to the root) as being either *active* or *redundant*. A redundant package can freely be overwritten, but not an active package.

If x is an up-package currently held by some process Q which is not the root, then x is redundant if x has already been copied by Q.*par*. If x is an up-package currently held by the root, then x is redundant if the root has already created a down-package with the same guide pair as x. Any other up-package is active.

If x is a down-package held by some process Q which is not its home process, then Q is redundant if it has been copied by some child of Q. (The child that copies x must be the process whose subtree contains the home process of x.) If x is a down-package held by its home process P, then x is redundant if P.*rank* is equal to the value of x. This indicates that P has already copied its rank from x, or that P.*rank* was correct before x arrived. Any other down-package is active.

Status Waves. As it is typical for distributed algorithms which are self-stabilizing, but not silent, CRK endlessly repeats the calculation of the ranks of the processes in $\mathcal{T}$. We call one (complete) pass through this cycle of computations an *epoch*. At the end of each epoch, the variables of CRK at all processes, other than the variables for weight and rank, are reset for the next epoch. If an epoch has a clean start, it will calculate the correct rank for each process. Subsequent epochs will simply recalculate the same value, and $P.rank$ will never change again.

On the other hand, in case of an arbitrary initial configuration, it is possible for incorrect values of rank to be calculated, but eventually a configuration will be reached when the next epoch will get a clean start.

This system is controlled by the *status* variables of the processes. At the beginning of an epoch, a broadcast wave starting from the root changes the status of every process from either 0 or 4 to 1, and all variables of CRK except rank and weight are set to their initial values. When this wave reaches the leaves of $\mathcal{T}$, a convergecast wave changes the status of all processes to 2. All computation of the ranking algorithm, as discussed above, takes place while processes have status 2. After the root has created the last down-package, it initiates a broadcast wave where the status of all processes changes to 3. The return convergecast wave then changes the status of all processes to 4, and when this wave reaches the root, the new epoch begins.

Status zero is used for error correction. If any process detects that the current epoch is erroneous, it changes its status to 0. Status 0 spreads down the tree, as well as up the tree unless it meets a process whose status is 1. If $Root.status$ becomes 0 (and all its children have status 0 or 4), then $Root$ initiates a status 1 broadcast wave starting a new epoch. However, this may cause an endless cycle of 0 and 1 wave, going up and down the tree, respectively. We solve this problem by adding a special rule for the non-root processes. If $P.status = 0$ and $P.par.status = 1$, the status 0 wave cannot move up; instead, the status 0 wave moves down followed by status 1 wave.

4.2 Formal Definition of CRK

Variables of CRK. Let P be any process. $P.par$, $P.guide$, and $P.weight$ are inputs of CRK. Then, the output of CRK is $P.rank$, an integer. To compute this output, P maintains the following additional variables:

1. $P.up\_pkg$ and $P.down\_pkg$ are respectively of package type (that is, a guide pair and an integer) or $\perp$ (undefined).
   If $P.up\_pkg$ (resp. $P.down\_pkg$) is defined, then its home process is some $Q \in \mathcal{T}_P$.
2. $P.started$, Boolean.
   This variable indicates whether P has already generated its up-package during this epoch. ($P.up\_pkg$ may or may not still contain that up-package.)
3. $P.up\_done$, Boolean.
   It indicates whether all processes in $\mathcal{T}_P$ have created their own up-package in the current epoch and whether $\mathcal{T}_P$ contains no active up-package. (Active up-packages whose home processes are in $\mathcal{T}_P$ could exist at processes above P.)
4. $P.status \in [0..4]$.
   Status variables are used to control the order of computation and to correct errors.

Finally, $Root$ contains the following additional variable:

5. $Root.counter \in \mathbb{N}$
   This incrementing integer variable assigns the rank to packages. It is initialized to be 0 every time a new epoch begins.

Predicates of CRK**.** The predicate *Clean_State*(P) below indicates if P is in a good initial or "clean" state.

$$\textit{Clean\_State}(P) \equiv P.\textit{up\_pkg} = \bot \wedge P.\textit{down\_pkg} = \bot \wedge \neg P.\textit{started} \wedge \neg P.\textit{up\_done}$$

The four following predicates are used for error detection:

$$\begin{aligned}
\textit{Is_Consistent}(P, g) &\equiv g = P.\textit{guide} \vee \exists Q \in \textit{Chldrn}(P), g \geq Q.\textit{guide} \\
\textit{Guide_Error}(P) &\equiv (P.\textit{up_pkg} \neq \bot \wedge \neg \textit{Is_Consistent}(P, P.\textit{up_pkg}.\textit{guide})) \vee \\
&\quad (P.\textit{down_pkg} \neq \bot \wedge \neg \textit{Is_Consistent}(P, P.\textit{down_pkg}.\textit{guide})) \\
\textit{Status_Error}(P) &\equiv (P.\textit{status} \in \{1,3\} \wedge P.\textit{par}.\textit{status} \neq P.\textit{status}) \vee \\
&\quad (P.\textit{status} \in \{2,4\} \wedge \exists Q \in \textit{Chldrn}(P), Q.\textit{status} \neq P.\textit{status}) \vee \\
&\quad (P.\textit{status} \neq 0 \wedge P.\textit{par}.\textit{status} = 0) \vee \\
&\quad (P.\textit{status} \notin \{0,1\} \wedge \exists Q \in \textit{Chldrn}(P), Q.\textit{status} = 0) \\
\textit{Error}(P) &\equiv \textit{Status_Error}(P) \vee \\
&\quad (\neg \textit{Clean_State}(P) \wedge P.\textit{status} = 1) \vee \\
&\quad (\textit{Guide_Error}(P) \wedge P.\textit{status} = 2) \vee \\
&\quad (P.\textit{up_done} \wedge \neg P.\textit{started} \wedge P.\textit{status} = 2) \vee \\
&\quad (P.\textit{up_done} \wedge P.\textit{status} = 2 \wedge \exists Q \in \textit{Chldrn}(P), \neg Q.\textit{up_done})
\end{aligned}$$

We say that a guide pair g is *consistent with* P if the predicate *Is_Consistent*(P, g) is true. If *Is_Consistent*(P, g) is false, g is the guide pair of no process in the subtree of P. *Guide_Error*$(P) = true$ means that P holds a package whose home is not in the subtree of P. The predicate *Status_Error*(P) indicates whether P detects that its status is inconsistent with those of its neighbors. Status errors are always the result of arbitrary initializations; eventually, *Status_Error*(P) will become false and will remain false forever for all P. Finally, the predicate *Error*(P) detects error in the context of the current wave.

The four following predicates are used for flow control:

$$\begin{aligned}
\textit{Up_Redundant}(P) &\equiv (P \neq Root \wedge P.\textit{up_pkg} \neq \bot \wedge P.\textit{par}.\textit{up_pkg} \neq \bot \wedge P.\textit{par}.\textit{up_pkg} \geq P.\textit{up_pkg}) \vee \\
&\quad (P = Root \wedge P.\textit{up_pkg} \neq \bot \wedge P.\textit{down_pkg} \neq \bot \wedge P.\textit{down_pkg}.\textit{guide} = P.\textit{up_pkg}.\textit{guide}) \\
\textit{Down_Ready}(P) &\equiv P.\textit{down_pkg} = \bot \vee (P.\textit{down_pkg} \neq \bot \wedge \\
&\quad (P.\textit{down_pkg}.\textit{guide} \neq P.\textit{guide} \wedge \exists Q \in \textit{Chldrn}(P), Q.\textit{down_pkg} = P.\textit{down_pkg}) \vee \\
&\quad (P.\textit{down_pkg}.\textit{guide} = P.\textit{guide} \wedge P.\textit{rank} = P.\textit{down_pkg}.value)) \\
\textit{Can_Start}(P) &\equiv \neg P.\textit{started} \wedge (P.\textit{up_pkg} = \bot \vee \textit{Up_Redundant}(P)) \wedge \forall Q \in \textit{Chldrn}(P), \\
&\quad (\neg \textit{Up_Redundant}(Q) \vee Q.\textit{up_done}) \wedge (Q.\textit{up_pkg} > (P.\textit{weight}, P.\textit{guide}) \vee Q.\textit{up_done}) \\
\textit{Can_Copy_Up}(P, Q) &\equiv Q \in \textit{Chldrn}(P) \wedge (Q.\textit{up_pkg} \neq \bot \wedge \neg \textit{Up_Redundant}(Q)) \wedge \\
&\quad (P.\textit{up_pkg} = \bot \vee \textit{Up_Redundant}(P)) \wedge \\
&\quad (P.\textit{started} \vee (P.\textit{weight}, P.\textit{guide}) > Q.\textit{up_pkg}) \wedge \forall R \in \textit{Chldrn}(P), \\
&\quad R.\textit{up_done} \vee (\neg \textit{Up_Redundant}(R) \wedge (R.\textit{up_pkg} \geq Q.\textit{up_pkg} \vee R.\textit{up_done})
\end{aligned}$$

P.*up_pkg* is redundant if *Up_Redundant*(P) is true. *Down_Ready*(P) states whether P.*down_pkg* is redundant or undefined, and thus P can create or copy a new down-package. *Can_Start*(P) decides whether P can create its own package, that is, if P can set P.*up_pkg* to $(P$.*weight*, P.*guide*$)$. *Can_Copy_Up*(P) indicates whether P can copy Q.*up_pkg* to P.*up_pkg*. We note that P can evaluate *Up_Redundant*(Q) for any $Q \in$ *Chldrn*(P).

Predicate *Up_Done*(P) below decides whether all processes in $\mathcal{T}_P$ have created their own up-package in the current epoch and whether $\mathcal{T}_P$ contains no active up-package. The evaluation of *Up_Done*(P) gives the correct value for P.*up_done*.

$$\textit{Up\_Done}(P) \equiv P.\textit{started} = true \ \wedge \textit{Up\_Redundant}(P) \ \wedge \forall Q \in \textit{Chldrn}(P), Q.\textit{up\_done}$$

Actions of CRK**.** Actions of CRK are given below. To simplify the design, we assume that the actions of CRK use the same priorities as those of CGP.

For the root process $Root$ **only:**

`Err`	:: *Error*$(Root)$	$\mapsto$ $Root$.*status* $\leftarrow 0$
`NewEpoch`	:: $Root$.*status* $\in \{0,4\} \wedge$ $\forall Q \in$ *Chldrn*$(Root)$, Q.*status* $\in \{0,4\}$	$\mapsto$ $Root$.*status* $\leftarrow 1$; *counter* $\leftarrow 0$ $Root$.*up_pkg* $\leftarrow \perp$; $Root$.*down_pkg* $\leftarrow \perp$ $Root$.*started* $\leftarrow false$; $Root$.*up_done* $\leftarrow false$
`ConvCast`	:: $Root$.*status* $= 1 \wedge$ $\forall Q \in$ *Chldrn*$(Root), Q$.*status* $= 2$	$\mapsto$ $Root$.*status* $\leftarrow 2$
`CreateUpPkg`	:: $Root$.*status* $= 2 \ \wedge$ *Can_Start*$(Root)$	$\mapsto$ $Root$.*up_pkg.value* $\leftarrow$ $Root$.*weight* $Root$.*up_pkg.guide* $\leftarrow$ $Root$.*guide* $Root$.*started* $\leftarrow true$
`CopyUpPkg`	:: $Root$.*status* $= 2 \wedge$ $\exists Q \in$ *Chldrn*$(Root)$, *Can_Copy_Up*$(Root, Q)$	$\mapsto$ $Root$.*up_pkg* $\leftarrow Q$.*up_pkg*, $Q = min_{\prec_{Root}}\{R \in$ *Chldrn*$(Root)$, *Can_Copy_Up*$(Root, R)\}$
`EndUpPkg`	:: $Root$.*started* $\wedge$ *Up_Redundant*$(Root) \wedge$ $\forall Q \in$ *Chldrn*$(Root), Q$.*up_done*	$\mapsto$ $Root$.*up_done* $\leftarrow true$
`CreateDownPkg`	:: *Down_Ready*$(Root) \wedge$ $Root$.*up_pkg* $\neq \perp \wedge$ $\neg$*Up_Redundant*$(Root)$	$\mapsto$ *counter* $\leftarrow$ *counter* $+ 1$ $Root$.*down_pkg.value* $\leftarrow$ *counter* $Root$.*down_pkg.guide* $\leftarrow$ $Root$.*up_pkg.guide*
`SetRank`	:: $Root$.*down_pkg* $\neq \perp \wedge$ $Root$.*down_pkg.guide* $=$ $Root$.*guide* $\wedge$ $Root$.*down_pkg.value* $\neq$ $Root$.*rank*	$\mapsto$ $Root$.*rank* $\leftarrow$ $Root$.*down_pkg.value*
`BroadCast`	:: $Root$.*status* $= 2 \wedge$ $Root$.*up_done* $\wedge$ *Down_Ready*$(Root)$	$\mapsto$ $Root$.*status* $\leftarrow 3$
`EndEpoch`	:: $Root$.*status* $= 3 \wedge$ $\forall Q \in$ *Chldrn*$(Root), Q$.*status* $= 4$	$\mapsto$ $Root$.*status* $\leftarrow 4$

For every non-root process P only:

`Err`	::	$\mathit{Error}(P)$	$\mapsto$ $P.\mathit{status} \leftarrow 0$
`NewEpoch`	::	$P.\mathit{par}.\mathit{status} = 1 \ \wedge P.\mathit{status} \in \{0,4\} \wedge$ $\forall Q \in \mathit{Chldrn}(P), Q.\mathit{status} \in \{0,4\}$	$\mapsto$ $P.\mathit{status} \leftarrow 1$ $P.\mathit{up\_pkg} \leftarrow \perp$ $P.\mathit{down\_pkg} \leftarrow \perp$ $P.\mathit{started} \leftarrow \mathit{false}$ $P.\mathit{up\_done} \leftarrow \mathit{false}$
`ConvCast`	::	$P.\mathit{status} = 1 \ \wedge \forall Q \in \mathit{Chldrn}(P), Q.\mathit{status} = 2$	$\mapsto$ $P.\mathit{status} \leftarrow 2$
`CreateUpPkg`	::	$P.\mathit{status} = 2 \ \wedge \mathit{Can\_Start}(P)$	$\mapsto$ $P.\mathit{up\_pkg}.\mathit{value} \leftarrow P.\mathit{weight}$ $P.\mathit{up\_pkg}.\mathit{guide} \leftarrow P.\mathit{guide}$ $P.\mathit{started} \leftarrow \mathit{true}$
`CopyUpPkg`	::	$P.\mathit{status} = 2 \wedge$ $\exists Q \in \mathit{Chldrn}(P), \mathit{Can\_Copy\_Up}(P,Q)$	$\mapsto$ $P.\mathit{up\_pkg} \leftarrow Q.\mathit{up\_pkg}$, $Q = \min_{\prec_P}\{R \in \mathit{Chldrn}(P),$ $\mathit{Can\_Copy\_Up}(P,R)\}$
`EndUpPkg`	::	$P.\mathit{started} \ \wedge \mathit{Up\_Redundant}(P) \wedge$ $\forall Q \in \mathit{Chldrn}(P), Q.\mathit{up\_done}$	$\mapsto$ $P.\mathit{up\_done} \leftarrow \mathit{true}$
`CopyDownPkg`	::	$\mathit{Down\_Ready}(P) \wedge$ $P.\mathit{par}.\mathit{down\_pkg} \neq \perp \wedge$ $P.\mathit{par}.\mathit{down\_pkg} \neq P.\mathit{down\_pkg} \wedge$ $\mathit{Is\_Consistent}(P, P.\mathit{par}.\mathit{down\_pkg})$	$\mapsto$ $P.\mathit{down\_pkg} \leftarrow P.\mathit{par}.\mathit{down\_pkg}$
`SetRank`	::	$P.\mathit{down\_pkg} \neq \perp \wedge$ $P.\mathit{down\_pkg}.\mathit{guide} = P.\mathit{guide} \wedge$ $P.\mathit{down\_pkg}.\mathit{value} \neq P.\mathit{rank}$	$\mapsto$ $P.\mathit{rank} \leftarrow P.\mathit{down\_pkg}.\mathit{value}$
`BroadCast`	::	$P.\mathit{par}.\mathit{status} = 3 \ \wedge P.\mathit{status} = 2 \wedge$ $\forall Q \in \mathit{Chldrn}(P), Q.\mathit{status} = 2 \ \wedge \mathit{Down\_Ready}(P)$	$\mapsto$ $P.\mathit{status} \leftarrow 3$
`EndEpoch`	::	$P.\mathit{status} = 3 \ \wedge \forall Q \in \mathit{Chldrn}(P), Q.\mathit{status} = 4$	$\mapsto$ $P.\mathit{status} \leftarrow 4$

The actions above achieve three tasks. They are (1) error correction, (2) epochs, and (3) ranking computation (using the flow packages).

Error Correction. Action `Err` performs the error correction. If one process detects any inconsistency among its state and that of its neighbors, it initiates a reset of the network by changing its status to 0. This reset is contagious as previously explained.

Epochs. A new epoch starts by a reset initiated by Action `NewEpoch` at the root: If $\mathit{Root}.\mathit{status}$ is either 0 or 4, and every child of Root has status 0 or 4, then Root broadcasts the status 1 and resets to a clean state.

When status 1 reaches the leaves, a convergecast wave starts and changes the status of all processes to 2 by Action `ConvCast`, so that actual ranks computation can begin.

When Root detects that there are no more up-packages in the tree, and it already sent every down-package, it initializes a broadcast of status 3 by Action `BroadCast`. Note that there could still be active down-packages below Root, but there could not be any active up-packages. Thus, Root is finished with its tasks for the current epoch. Non-root process P propagates the status 3 by Action `BroadCast` after sending all its down-packages. There could still be active down-packages below P, but no active up-packages. Since $P.\mathit{par}.\mathit{status} = 3$, P knows that its job for this epoch is done.

Once status 3 reaches the leaves, a convergecast of status 4 is initialized and propagated by Action `EndEpoch`. When Root changes to status 4, the current epoch is done, and Root initiates a new one.

Ranking computation. The computation of the ranking is bottom-up and starts when the convergecast of status 2 starts at the leaves. The flow of up-packages is organized using `CreateUpPkg` and `CopyUpPkg`, that is, a process either inserts its own package in the flow or copy some package coming from a child by ensuring that packages are moved up in ascending order of weight. Once a process P has detected that $\mathcal{T}_P$ has no active up-package, it sets P.*up_done* to true by Action `EndUpPkg`. $Root$ initializes the broadcast of status 3 only after $Root$.*up_done* switches to true.

Upon receiving a new up-package (that is, $Root$.*up_pkg* is active), if $Root$.*down_pkg* is available (that is, it is either $\perp$ or redundant), $Root$ is enabled to create a new down-package to send down to the home of its up-package by `CreateDownPkg`. If *counter* $= i$, then $Root$.*up_pkg* is the i^{th} up-package copied or created by $Root$, its weight is the i^{th} smallest weight in the network, and i will become the value of the down-package.

The new active down-package is propagated to its home process by successive copying using Action `CopyDownPkg`. When it reaches its home process P, the value field of that package contains the correct value of the rank of P, so P updates $P.rank$ using Action `SetRank`, if necessary.

Theorem 3. RANK *is self-stabilizing, computes the ranking of all processes in* $O(n)$ *rounds from an arbitrary initial configuration, and works under the weakly fair scheduler.*

References

1. Bein, D., Datta, A., Villain, V.: Snap-stabilizing optimal binary-search-tree. Proceedings of the 7-th International Symposium on Self-Stabilizing Systems (2005)
2. Bourgon, B., Datta, A.K., Natarajan, V.: A self-stabilizing ranking algorithm for tree structured networks. In: Proceedings of the First Workshop on Self-Stabilizing Systems (WSS'95). pp. 23–28 (1995)
3. Datta, A.K., Devismes, S., Heurtefeux, K., Larmore, L.L., Rivierre, Y.: Self-stabilizing small k-dominating sets. Tech. rep., VERIMAG (2011), `http://www-verimag.imag.fr/TR/TR-2011-6.pdf`
4. Dijkstra, E.: Self stabilizing systems in spite of distributed control. Communications of the Association of Computing Machinery 17, 643–644 (1974)
5. Dolev, S.: Self-Stabilization. The MIT Press (2000)
6. Dolev, S., Gouda, M., Schneider, M.: Memory requirements for silent stabilization. In: PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing. pp. 27–34 (1996)
7. Flocchini, P., Enriques, A.M., Pagli, L., Prencipe, G., Santoro, N.: Point-of-failure shortest-path rerouting: Computing the optimal swap edges distributively. IEICE Transactions 89-D(2), 700–708 (2006)
8. Herman, T., Masuzawa, T.: A stabilizing search tree with availability properties. In: Fifth International Symposium on Autonomous Decentralized Systems (ISADS 2001). pp. 398–405 (2001)
9. Herman, T., Pirwani, I.: A composite stabilizing data structure. 5th International Workshop on Self-Stabilizing Systems (WSS 2001), Lecture Notes in Computer Science LNCS 2194, Springer Verlag pp. 167–182 (2001)
10. Tel, G.: Introduction to distributed algorithms (2nd Ed.). Cambridge University Press (2000)