Annales UMCS Informatica AI XIII, 1 (2013) 7–10
DOI: 10.2478/v10065-012-0038-7

Annales UMCS
Informatica
Lublin-Polonia
Sectio AI
http://www.annales.umcs.lublin.pl/

A self-stabilizing algorithm for detecting fundamental cycles in a graph with DFS spanning tree given

Halina Bielak[1]*, Michał Pańczyk[2]†

[1] *Institute of Mathematics, Maria Curie-Sklodowska University, pl. M. Curie-Sklodowskiej 5, 20-031 Lublin, Poland*
[2] *Institute of Computer Science, Maria Curie-Sklodowska University, pl. M. Curie-Sklodowskiej 5, 20-031 Lublin, Poland*

Abstract – This paper presents a linear time self-stabilizing algorithm for detecting the set of fundamental cycles on an undirected connected graph modelling asynchronous distributed system. The previous known algorithm has $O(n^2)$ time complexity, whereas we prove that this one stabilizes after $O(n)$ moves. The distributed adversarial scheduler is considered. Both algorithms assume that the depth-search spanning tree of the graph is given. The output is given in a distributed manner as a state of variables in the nodes.

1 Introduction

A notion of self-stabilizing algorithms on the distributed systems was introduced by Dijkstra [**1**] in 1974. They can model distributed systems with message passing or shared memory. A survey in the topic can be found in the paper by Schneider [**2**], and more details in the book by Dolev [**3**]. The notions from the graph theory not defined in this paper can be found in the book by Harary [**4**].

A distributed self-stabilizing system consists of a set of processes (i.e. computing nodes) and communication links between them. Every node in the system runs the same algorithm and can change the state of local variables. These variables determine *local state* of a node. Nodes can observe the state of variables on themselves and their neighbour nodes. The state of all the nodes in the system determines the *global state*.

*hbiel@hektor.umcs.lublin.pl
†mjpanczyk@gmail.com

Every self-stabilizing algorithm should have a class of global states called *legitimate state* defined, when the system is stable and no action can be done by the algorithm itself. Every other global state is called *illegitimate* and for the algorithm to be correct there has to be some possibility to make a move if the state is illegitimate. The aim of the self-stabilizing algorithm is to bring the legitimate (desirable) state of the whole system after some alteration (from the outside of the system) of variables in the nodes or after the system has been started.

In this paper we present an $O(n)$ time modification (we call it ASFC II) of Chaudhuri's **[5]** algorithm (ASFC I) for detecting fundamental cycles in a graph. A set of fundamental cycles (SFC) is a collection of cycles that can be used to construct any other cycle in the graph. Given any cycle C from graph G, it can be constructed as $C = C_1 \oplus C_2 \oplus \ldots \oplus C_k$, where $C_i \in SFC(G)$ and $G_1 \oplus G_2$ is such a subgraph constructed from the subgraphs G_1 and G_2 of the G graph, that each of its edges exists either in G_1 or G_2, so this is the symmetric difference.

ASFC I requires $O(n^2)$ time to stabilize, where n is the number of processes in a system. It is able to start at arbitrary configuration, given that the DFST spanning tree is known. In fact, $O(n^2)$ time complexity was assumed when no information about the spanning tree structure can be altered. The information about the spanning tree is meant to be stored in a protected memory.

ASFC II requires the additional 2-bit variable q in each process. The initial state of q should be set right after (or during) finding the DFS spanning tree. It must be also stored in a protected memory. After finding DFST and setting q, the essential part of the algorithm can be started (see Chapter 3 for the details).

The effects of running both algorithms are stored in a distributed manner as a state of local variables in the processing nodes; i.e. when a system running the algorithm stabilizes, each of its nodes knows how many fundamental cycles pass through it and what their identifiers are.

The rest of this article is organized as follows. Section 2 introduces the notation and model of computation used in next sections. Section 3 presents the original Chaudhuri algorithm and our improved version. In section 4 we prove convergence and time characteristic of our algorithm, and in the last section 5 we conclude the paper.

2 Notation and computational model

We consider a self-stabilizing system modelled by a finite, undirected graph $G = (V, E)$. Let the number of vertices be $n = |V|$ and the number of edges $e = |E|$. We think of every process as a node in the graph, whereas connection links between them are edges. There are some variables in each node. Names and types of the variables are set during design of an algorithm. Every node can also look up for the state of the variables at its neighbours.

Each node runs the same algorithm. The algorithm consists of a set of rules. A rule has the form as below.

Algorithm 1: An example rule

label: **if** *guard* **then**
assignment instructions

A *guard* is a logic predicate which can refer to variables in the node itself and its neighbours. We say that a rule is *active* if its guard is evaluated to be true. A node is *active* if it contains any active rule. If there is no active node in the graph, we say that the system is *stabilized*.

A self-stabilizing system contains also a *scheduler*. Its task is to choose one process from the set of active processes and to trigger an active rule in it. We call such an action a *move*. In this article we assume that the scheduler is distributed and adversarial, so the order of activating nodes is nondeterministic. As a consequence, while describing pessimistic complexity of the algorithm (number of moves), we must take the worst case scenario of triggering actions in particular nodes.

Now we present some notation used in the paper. Most of the following notations is the same as in [**5**]. As mentioned before, we consider the connected, undirected graph $G = (V, E)$ with the vertex set V and the edge set E. We also assume that the DFST spanning tree $T(r) = (V, E')$ with the root node r is given. Note that the set of nontree edges is $E - E'$. So we have the following notation:

$n(i)$**:** the set of neighbour nodes of the node i,
$p(i)$**:** the parent node of i in $T(r)$ (we assume that $p(r) = null$),
$c(i)$**:** the set of children of the node i in $T(r)$ (leaf nodes have $c(i) = \emptyset$),
$nt(i)$**:** the set of nontree edges incident to the node i ($nt(i) = \{\{i, k\} : \{i, k\} \notin E'\}$),
$C(i, j)$**:** the fundamental cycle created by the nontree edge $(i, j) \in E - E'$ together with the path between i and j in the tree $T(r)$,
$a(i)$**:** the set of ancestors of the node i (we assume that $i \in a(i)$),
$d(i)$**:** the set of descendants of the node i (we assume that $i \in d(i)$),
$s(i)$**:** the set of all nontree edges such that each of them connects a descendant and a proper ancestor of i (more precisely $s(i) = \{\{x, y\} : \{x, y\} \in E - E', x \in d(i), y \in a(i) - i\}$),
$su(i)$**:** $\bigcup_{j \in c(i)} s(j)$,
$fc(i)$**:** the set of nontree edges such that the fundamental cycle created by each of them passes through i (more precisely $fc(i) = \{\{x, y\} : \{x, y\} \in E - E', x \in a(i), y \in d(i)\}$),
$A \triangle B$**:** symmetric difference of the sets A and B ($A \triangle B = (A \cup B) - (A \cap B)$).

Some examples are presented in Fig. 1.

3 The algorithm

As it was mentioned before, our algorithm (ASFC II) is a modification of Chaudhuri's original one (ASFC I). Our change alters only the starting precondition for the system

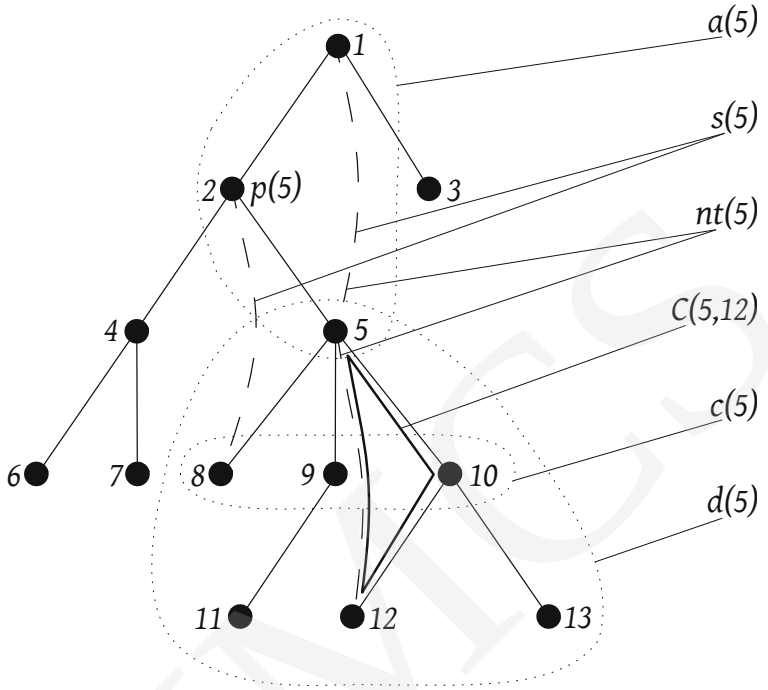

Fig. 1. Illustration of the notation for the node $i = 5$: $n(5) = \{1, 2, 8, 9, 10, 12\}$, $su(5) = \{\{2, 8\}, \{5, 12\}\}$, $fc(5) = \{\{1, 5\}, \{2, 8\}, \{5, 12\}\}$. Dashed lines indicate the nontree edges. Solid lines indicate the tree edges.

and the order in which processing nodes are triggered (so the complexity changes). The difference is that ASFC I runs in $O(n^2)$ time if there is the DFS spanning tree (DFST) of the system given, and in $O(n^3)$ time if DFST has not been detected yet. This situation requires running an algorithm for finding DFST. The algorithm developed by Colin and Dolev [**6**] can be used for that.

In contrast, our ASFC II algorithm runs in $O(n)$ time, but there has to be DFST given for the system and all the nodes must have the special variable $q(i)$ set to the **Stationary** state. If the DFST was not given or the information was corrupted, complexity is also $O(n^3)$ because of need to run the algorithm finding DFST.

When DFST is correct, but $q(i)$ is wrong in some nodes, complexity is $O(n^2)$. That is because pessimistic scenario is quite the same as in ASFC I, and $q(i)$ can be ignored while estimating the number of moves.

All the semantics connected essentially with SFC, i.e. states of variables representing SFC in the distributed manner in ASFC II are unaltered when compared to ASFC I. Thus we present here summary of the original algorithm; the complete version can be found in [**5**].

Let us assume DFST of the system is given, otherwise both algorithms must first calculate it. Two variables $p(i)$, $c(i)$ in each node state the structure of the tree and inferred from these two the value $nt(i)$.

Actually the main objective is to find $fc(i)$ for every node i in the system. The correct state of $fc(i)$ propagates along with $s(i)$ from leaves of the tree through internal nodes up to the root. The original Chaudhuri's ASFC I algorithm is given below.

Algorithm 2: The ASFC I algorithm

leaf: **if** $c(i) = \emptyset \wedge (s(i) \neq nt(i) \vee fc(i) \neq nt(i))$ **then**
  $s(i) := nt(i)$
  $fc(i) := nt(i)$
internal: **if** $c(i) \neq \emptyset \wedge (s(i) \neq nt(i) \triangle su(i) \vee fc(i) \neq su(i))$ **then**
  $s(i) := nt(i) \triangle su(i)$
  $fc(i) := su(i)$

Now we state the theorem whose proof can be found in [**5**].

Theorem 1. The ASFC I algorithm stabilizes the system after at most $O(n^2)$ moves, under the condition that DFS spanning tree is given.

The meaning of symmetric difference in the internal rule is that as the computation passes along the treepath from leaves to the root, the rule first incorporates the nontree edge to the set $s(i)$. The next time the same nontree edge is encountered (but the other end vertex of it), it is deleted from the $s(i)$ set.

A simple example of computation of the ASFC I algorithm, which exploits fully $O(n^2)$ time is when DFS search gives simple path and each node is active but every time the scheduler chooses a node nearest to the root. The first move would be done by the root node. The second move would be done by the unique child of the root, and then the third by the root because it was made active by triggering of its child. So in this way, the last phase would start from the unique leaf and finish in the root. There would be n phases, every i-th phase would involve i moves, so it gives $(n^2+n)/2$ moves.

In the ASFC II algorithm the idea is not to make correction of the node's variables $fc(i)$ or $s(i)$ immediately after the fault has occurred. It would cause a situation where ancestor nodes get "repaired" according to the incorrect information (not updated yet) from its children, as in the example above. Instead of this, every node is marked for updating and the update is done only if all the descendants are updated so they have correct values of $s(i)$ and $fc(i)$. A node cannot get back to the **Stationary** state right after updating correct values of $s(i)$ or $fc(i)$ because it has to wait for all the nodes to be corrected (see rule 2.b). After all the nodes get updated, every node is being marked as almost stationary (root node first and then all its descendants recursively). Then all nodes can turn into the stationary state — starting from leaves up to the root. An additional variable $q(i)$ for each node i is used

$$q(i) \in \{\textbf{Stationary}, \textbf{NeedUpdate}, \textbf{Updated}, \textbf{NearStationary}\}$$

(shortly: S, NU, U, NS), which represents the state of the node.

As it was mentioned above, we assume the DFST for the system is given and represented in a distributed way. According to Arora and Gouda [**7**] and Schneider [**2**], it is possible to make a precondition for a self-stabilizing system, therefore we assume a precondition such that the value of variable $q(i)$ for every node i is **Stationary**.

One can fulfill this requirement for example by putting the $q(i)$ variable in some kind of protected memory, e.g. such that this variable can be altered only by the algorithm — not by a change from outside of the system.

The ASFC II algorithm is given below.

Algorithm 3: The ASFC II algorithm

1.a): **if** $q(i) = \textbf{\textit{Stationary}} \wedge c(i) = \emptyset \wedge (s(i) \neq nt(i) \vee fc(i) \neq nt(i))$ **then**
$q(i) :=$ **NeedUpdate**

1.b): **if** $q(i) = \textbf{\textit{Stationary}} \wedge c(i) \neq \emptyset \wedge (s(i) \neq nt(i) \triangle su(i) \vee fc(i) \neq su(i))$ **then**
$q(i) :=$ **NeedUpdate**

1.c): **if** $q(i) = \textbf{\textit{Stationary}} \wedge q(p(i)) = \textbf{\textit{NeedUpdate}}$ **then**
$q(i) :=$ **NeedUpdate**

1.d): **if** $q(i) = \textbf{\textit{Stationary}} \wedge c(i) \neq \emptyset \wedge \exists_{k \in c(i)}(q(k) \in \{\textbf{\textit{NeedUpdate}}, \textbf{\textit{Updated}}\})$ **then**
$q(i) :=$ **NeedUpdate**

2.a): **if** $q(i) = \textbf{\textit{NeedUpdate}} \wedge c(i) = \emptyset$ **then**
$q(i) :=$ **Updated**
$s(i) := nt(i)$
$fc(i) := nt(i)$

2.b): **if** $q(i) = \textbf{\textit{NeedUpdate}} \wedge c(i) \neq \emptyset \wedge \forall_{k \in c(i)}(q(k) = \textbf{\textit{Updated}})$ **then**
$q(i) :=$ **Updated**
$s(i) := nt(i) \triangle su(i)$
$fc(i) := su(i)$

3.a): **if** $q(i) = \textbf{\textit{Updated}} \wedge p(i) = null \wedge \forall_{k \in c(i)}(q(k) = \textbf{\textit{Updated}})$ **then**
$q(i) :=$ **NearStationary**

3.b): **if** $q(i) = \textbf{\textit{Updated}} \wedge q(p(i)) = \textbf{\textit{NearStationary}}$ **then**
$q(i) :=$ **NearStationary**

4): **if** $q(i) = \textbf{\textit{NearStationary}} \wedge \forall_{k \in c(i)}(q(k) = \textbf{\textit{Stationary}})$ **then**
$q(i) :=$ **Stationary**

An illustration of the algorithm rules is presented in Fig. 2 where only tree edges are shown. Fig. 3 presents an examplary execution of the algorithm.

4 Convergence and complexity

We will show that the system stabilizes after exactly $4n$ moves if there are any faults regardless of their number. All the time we assume the precondition $\forall_{i \in V(G)} q(i) =$

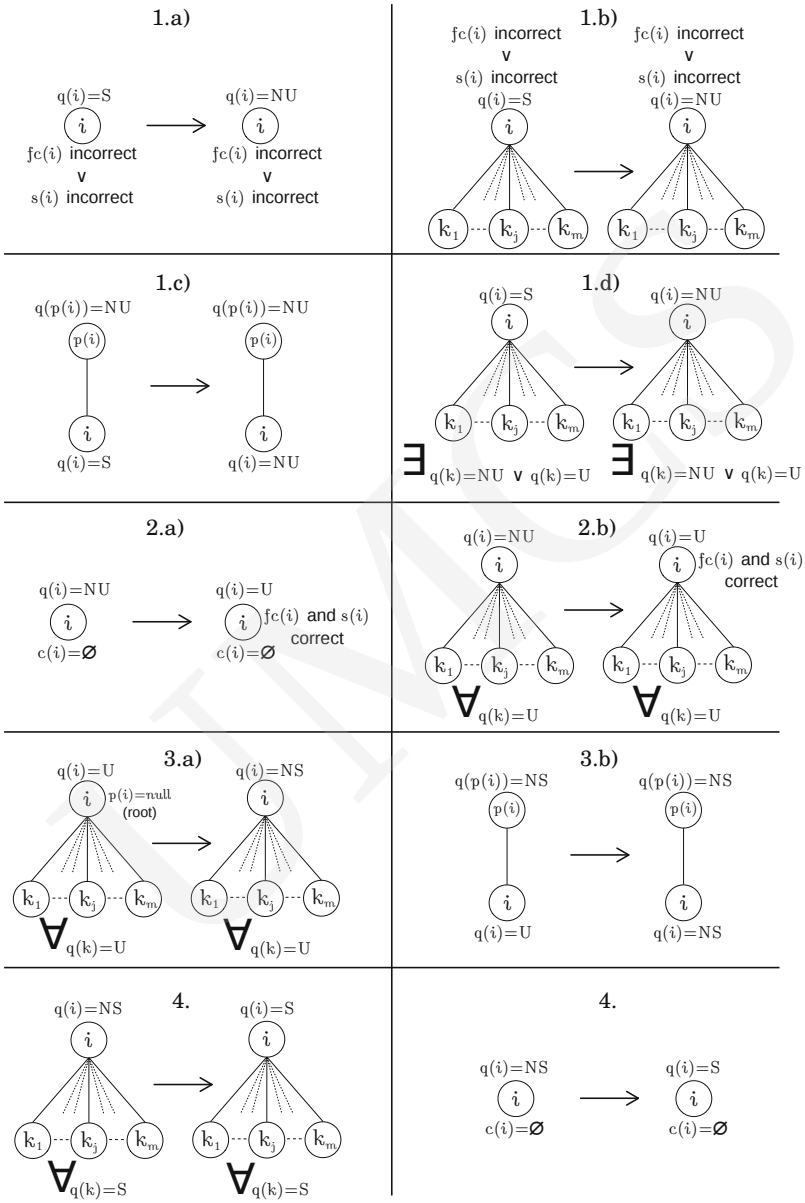

Fig. 2. An illustration of the ASFC II algorithm rules, where $k \in c(i)$.

Stationary is fulfilled after any transient fault has occurred and the DFST of the graph is also given.

Lemma 1. Each node i changes its state $q(i)$ cyclically from **Stationary** through **NeedUpdate**, **Updated**, **NearStationary**, back to **Stationary**.

PROOF. It follows from the rules of the algorithm. Only the rules 1.a)–1.d) change the state from $q(i) =$ **Stationary** and set it into the state $q(i) =$ **NeedUpdate**. Next, only rules 2.a) and 2.b) change the state from $q(i) =$ **NeedUpdate** and set it into the state $q(i) =$ **Updated**. Analogously, there are rules 3.a) and 3.b) changing $q(i) =$ **Updated** only into $q(i) =$ **NearStationary** and rule 4. changing $q(i) =$ **NearStationary** into $q(i) =$ **Stationary**. □

Lemma 2. If a node i changes its state $q(i)$ from **NeedUpdate** to **Updated**, then its each descendant has the correct state of $s(i)$, $fc(i)$ and $q(i) =$ **Updated**.

PROOF. According to rule 2.b), the non-leaf node i can change its state $q(i)$ from **NeedUpdate** to **Updated** only if its each child has the state $q(i) =$ **Updated**. For each child we can repeat our reasoning as for their parent. Going this way down to the leaves, we can see that every descendant has been in the state $q(i) =$ **Updated** at least for a while between the node i was **Stationary** and became **Updated**.

To end the proof, it is now sufficient to show that a node cannot switch itself from $q(i) =$ **Updated** if its parent also has the state $q(p(i)) =$ **Updated**. In other words, the node is inactive until its parent's state switches into $q(p(i)) =$ **NearStationary**. It follows from the fact that rule 3.b) is the unique rule that could change the state of the node i. But this rule cannot be active, since $q(p(i)) =$ **Updated**.

From the fact that the values of $s(i)$ and $fc(i)$ are set during switching to the state $q(i) =$ **Updated**, and this state propagates from leaves to the top up, implies that $s(i)$ and $fc(i)$ are correct.

□

From Lemma 2 we have that if the root node changes its state to **Updated**, every other node i in the tree has updated its states $s(i)$ and $fc(i)$ to its proper values and has set the state $q(i)$ to **Updated**.

Lemma 3. If a node is changing its state into $q(i) =$ **NearStationary**, then its each ascendant also has the state **NearStationary**.

PROOF. A node i must have parent's state $q(p(i)) =$ **NearStationary**, to change its state into $q(i) =$ **NearStationary** (rule 3.b). Repeating this reasoning inductively, we have that every ascendant had the state **NearStationary** at least in the past.

Now it is sufficient to show that it could not have changed it. For the change from **NearStationary** into **Stationary**, every child of a node has to be also in the **Stationary** state (rule 4). Again, repeating this reasoning down to the node i we have contrary to the fact that the node i is right about to change into the state **NearStationary** from **Updated**.

□

In particular by Lemma 3 we have that a leaf gets into the **NearStationary** state if its each ascendant also has the **NearStationary** state. Rule 4 makes a stoppage — a

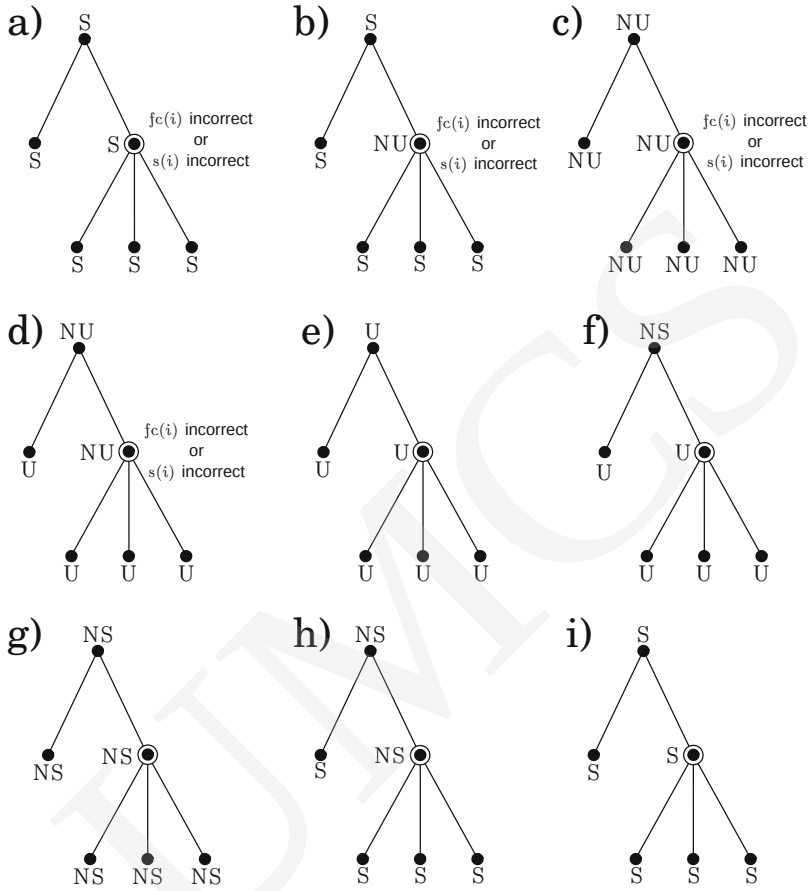

Fig. 3. An example of execution of the ASFC II algorithm:
a) there is one node i (circled in the figure) with the incorrect value of $fc(i)$ or $s(i)$,
b) its state $q(i)$ changes to **NeedUpdate** (rule 1.b),
c) its ancestors (rule 1.d) and descendants (rule 1.c) also get the **NeedUpdate** state,
d) leaves get the **Updated** state (rule 2.a),
e) and it propagates through the internal nodes up to the root (rule 2.b),
f) then root gets the **NearStationary** state (rule 3.a),
g) which propagates down to the leaves (rule 3.b),
h) then leaves get **Stationary** (rule 4),
i) and it propagates up to the root — now the system is stabilized (rule 4).

node has to wait for all its children to change from **NearStationary** into **Stationary**, before it changes its state into **Stationary**. By induction we have that it holds not only for children, but for all descendants.

Once a node goes through the whole sequence of changing of the own state from **Stationary**, through **NeedUpdate**, **Updated**, **NearStationary**, back to **Stationary**, it has the correct values of $fc(i)$ and $s(i)$. From Lemmas 1–3 we have that after a fault has occurred in at least one node, every node in the system has to go through all the sequence of changing. So we have the following theorem:

Theorem 2. If there is DFST given in a system, every node has the state **Stationary** and there is a number of nodes with the incorrect value of fc or s, the ASFC II algorithm stabilizes the system after $4n$ moves.

5 Conclusion

We have presented the modification of Chaudhuri's algorithm for finding a set of fundamental cycles in a graph. It demands a particular precondition on the state of a system, but thanks to this, it offers linear time of the stabilization. No matter how many faults have occurred in the system, the ASFC II algorithm always performs $4n$ moves.

Motivation for the problem, apart from the theoretical point of view, is applicable in computer networks. If we model a computer network as a graph, then the number of cycles passing through a node is a measure of its reliability. The more cycles go through a node, the more link failures (incident to the node) can occur whereas the graph is still connected.

References

[1] Dijkstra E. W., Self-stabilizing in spite of distributed control, Communications of the ACM 17 (1974): 643.
[2] Schneider M., Self-Stabilization, ACM Computing Surveys 25(1) (1993).
[3] Dolev S., Self-stabilization, The MIT Press, 2000.
[4] Harary F., Graph Theory, Addison-Wesley, 1972.
[5] Chaudhuri P., A Self-Stabilizing Algorithm for Detecting Fundamental Cycles in a Graph, Journal of Computer and System Sciences 59 (1999): 84.
[6] Collin Z., Dolev S., Self-stabilizing depth-first search, Information Processing Letters 49 (1994): 297.
[7] Arora A., Gouda M. G., Closure and convergence: A foundation for fault-tolerant computing, Proceedings of the 22nd International Conference on Fault-Tolerant Computing Systems (1992).