Report Number: WUCSE-2012-56

2012

Self-Stabilization in the Distributed Systems of Finite State Machines

Abusayeed Saifullah

The notion of self-stabilization was first proposed by Dijkstra in 1974 in his classic paper. The paper defines a system as self-stabilizing if, starting at any, possibly illegitimate, state the system can automatically adjust itself to eventually converge to a legitimate state in finite amount of time and once in a legitimate state it will remain so unless it incurs a subsequent transient fault. Dijkstra limited his attention to a ring of finite-state machines and provided its solution for self-stabilization. In the years following his introduction, very few papers were published in this area. Once his proposal was recognized... Read complete abstract on page 2.

Self-Stabilization in the Distributed Systems of Finite State Machines

Abusayeed Saifullah

Complete Abstract:

The notion of self-stabilization was first proposed by Dijkstra in 1974 in his classic paper. The paper defines a system as self-stabilizing if, starting at any, possibly illegitimate, state the system can automatically adjust itself to eventually converge to a legitimate state in finite amount of time and once in a legitimate state it will remain so unless it incurs a subsequent transient fault. Dijkstra limited his attention to a ring of finite-state machines and provided its solution for self-stabilization. In the years following his introduction, very few papers were published in this area. Once his proposal was recognized as a milestone in work on fault tolerance, the notion propagated among the researchers rapidly and many researchers in the distributed systems diverted their attention to it. The investigation and use of self-stabilization as an approach to fault-tolerant behavior under a model of transient failures for distributed systems is now undergoing a renaissance. A good number of works pertaining to self-stabilization in the distributed systems were proposed in the yesteryears most of which are very recent. This report surveys all previous works available in the literature of self-stabilizing systems.

2012-56

Self-Stabilization in the Distributed Systems of Finite State Machines

Authors: Abusayeed Saifullah

Corresponding Author: saifullaha@cse.wustl.edu

Web Page: http://www.cse.wustl.edu/~saifullaha/

Abstract: The notion of self-stabilization was first proposed by Dijkstra in 1974 in his classic paper. The paper defines a system as self-stabilizing if, starting at any, possibly illegitimate, state the system can automatically adjust itself to eventually converge to a legitimate state in finite amount of time and once in a legitimate state it will remain so unless it incurs a subsequent transient fault. Dijkstra limited his attention to a ring of finite-state machines and provided its solution for self-stabilization. In the years following his introduction, very few papers were published in this area. Once his proposal was recognized as a milestone in work on fault tolerance, the notion propagated among the researchers rapidly and many researchers in the distributed systems diverted their attention to it. The investigation and use of self-stabilization as an approach to fault-tolerant behavior under a model of transient failures for distributed systems is now undergoing a renaissance. A good number of works pertaining to self-stabilization in the distributed systems were proposed in the yesteryears most of which are very recent. This report surveys all previous works available in the literature of self-stabilizing systems.

Type of Report: Other

Self-Stabilization in the Distributed Systems of Finite State Machines

Abusayeed Saifullah
Department of Computer Science and Engineering
Washington University in St Louis
saifullaha@cse.wustl.edu

Contents

List of Figures

List of Tables

List of Algorithms

Abstract

The notion of self-stabilization was first proposed by Dijkstra in 1974 in his classic paper [35]. The paper defines a system as self-stabilizing if, starting at any, possibly illegitimate, state the system can automatically adjust itself to eventually converge to a legitimate state in finite amount of time and once in a legitimate state it will remain so unless it incurs a subsequent transient fault. Dijkstra [35] limited his attention to a ring of finite-state machines and provided its solution for self-stabilization which he proved later in [36]. In the years following his introduction, very few papers were published in this area. Once his proposal was recognized as a milestone in work on fault tolerance, the notion propagated among the researchers rapidly and many researchers in the distributed systems diverted their attention to it. The investigation and use of self-stabilization as an approach to fault-tolerant behavior under a model of transient failures for distributed systems is now undergoing a renaissance.

A good number of works pertaining to self-stabilization in the distributed systems were proposed in the yesteryears most of which are very recent. This report summarizes almost all the previous works available in the literature of self-stabilizing systems. Among the recent works, the proposals of constructing spanning tree [24], breadth first tree [74], tree-structured system [87], maximal matching [73] etc. are most remarkable.

Chapter 1

Introduction

Dijkstra [35] distinguishes any system having the property that, regardless of the initial state, the system is guaranteed to reach a legitimate state in a finite number of steps without any outside intervention. Such a property is very desirable for any distributed system to fight against any unexpected perturbation and to return to legal state without outside intervention. A self-stabilizing system does not need to be initialized and can recover from transient failures. If the system is designed to tolerate the temporary violation of a system specification, then the initial state specification is not needed.

A system lacking the property of self-stabilization fail to confront the unexpected perturbation and may stay in an illegitimate state forever. Most of the phenomena that contribute to the unexpected perturbation of a distributed system are indistinguishable. A few of them are as follows [100]:

(i) **Inconsistent initialization:** The processes that comprise the system may be initialized to local states that are inconsistent with one another.

(ii) **Mode change:** If a system is designed to execute in different modes, it is impossible for all of the processes to effect the change at the same time while changing the mode of operation .

(iii) **Transmission errors:** An inconsistency between the states of sender and receiver may happen due to the loss, corruption, or reordering of messages.

(iv) **Process failure and recovery:** The local state of a process may be inconsistent with the rest of the system when a process returns to service after going down.

(v) **Memory crash:** Inconsistency in local state of a process with the rest of the system may happen due to its local memory crash.

Each of these issues was handled separately, one at a time, and yet these seemingly disparate failure phenomena all have a common antidote, that of the self-stabilizing system. The traditional incremental and ad hoc approach is analogous to the use of exception handlers for the purpose of fault-tolerant software. Each addition of an exception condition may indeed reduce the possibility of a failure, but without a formal basis its elimination can never be guaranteed . Self-stabilization is the sole solution of this frustration. It provides a formal and unified approach to fault tolerance with respect to a model of transient failures and makes the departure from previous approaches to fault tolerance.

Dijkstra observed the complication that the behavior of a machine could only be influenced by the part of the total system state description that was available in that machine. The local actions taken on account of local information of a machine must accomplish a global objective. Dijkstra's notion of self-stabilization had a narrow scope of application, but it is the idea to break the ice to encompass a formal and unified approach to fault tolerance under a model of transient failures for distributed systems.

All the works proposed in the recent years for the self-stabilizing systems followed the same basic idea of Dijkstra [35] with a little modifications in some works. For each machine one or more privileges are defined. Privileges are boolean functions of a machine's own state and the states of its neighbors. When such a boolean function is true it is said that there is a privilege. If more than one privileges exist at the same time a central daemon selects one of the privileges. A machine enjoying the privilege makes a move that takes the machine into a new state. A predicate is defined to test the global state of the system. If the predicate is true, we say that the system is in a legitimate state. The main contribution of any self-stabilizing algorithm is that it can take the system into a legitimate state after a finite number of moves.

In this report I survey all the works that were proposed in the literature of self-stabilizing systems in the previous years. Throughout the report I represent a distributed system by a graph $G = (V, E)$ where V is the set of vertices and E is the set of edges.The vertices represent the machines or processors and the edges represent their inter-connections. I have used the term machine and node interchangeably throughout the report.

Chapter 2 contains some basic ideas for understanding the self-stabilizing systems. Chapter 3 gives a general discussion on the topic. Chapter 4 explains the early research works and chapter 5 summarizes all the recent works. This is followed by a brief review of the research and possible future works in this area.

Chapter 2

Definitions and Basic Ideas

2.1 Distributed System

A distributed system is generally defined as a set of loosely connected processing elements or state machines which do not share a common or global memory. Each node maintains a set of local variables whose contents specify the local state of that node. Machines placed in directly connected nodes are called each other's neighbors. The behavior of each machine depends only on local information. A node can read its own state and the state of its neighbors. There is no global information available in a node. Based on the network topology and signal propagation delay, each node contributes only partially to the global state of the system. The global state refers to the union of the local states of all nodes in the system. Two classes of global states are defined for such a system, depending on some predefined global criteria, which are

(i) the legitimate state, and

(ii) the illegitimate state.

2.2 Self-stabilizing System

We define self-stabilization for a system S with respect to a predicate P, over its set of global states, where P is intended to identify its correct execution. S is self-stabilizing with respect to predicate P if it satisfies the following two properties:

(i) **Closure**: P is closed under the execution of S. That is, once P is established in S, it cannot be falsified.

(ii) **Convergence**: Starting from an arbitrary global state, S is guaranteed to reach a global state satisfying P within a finite number of state transitions.

States satisfying (not satisfflng) P are called legitimate (illegitimate) states respectively. Dijkstra [35] defined as legitimate those states meeting a global-correctness criterion with the following four additional constraints:

(i) In each legitimate state one or more privileges will be present.

(ii) In each legitimate state each possible move will bring the system again in a legitimate state.

(iii) Each privilege must be present in at least one legitimate state.

(iv) For any pair of legitimate states there exists a sequence of moves transferring the system from the one into the other.

However, depending on the system specifications and criteria of the problems, these requirements have been modified in some papers.

The goal in a self-stabilizing distributed system is to start from an arbitrary (possibly illegitimate) initial state and then to reach a legitimate state after a finite number of moves (steps). Self-stabilizing algorithms are resilient to transient faults that perturb the state of the system arbitrarily. That is, if unexpected perturbations bring the system from a legitimate state to an illegitimate state, then the system must be able to again reach a legitimate state after a finite number of moves without any external intervention.

2.3 Self-Stabilizing Algorithm

Algorithm 1 General self-stabilizing algorithm

```
if < predicate > then
    the system is in legitimate state
else
    if < privilege > then
        < corresponding move >
    end if
end if
```

Almost all self-stabilizing algorithms are encoded as a set of rules. Each rule has two parts: the privilege and the move part as explained in chapter 1. The privilege part is

defined as a boolean function of the processor's own state and the states of its neighbors. When the privilege part of a rule is true on a processor then that processor enjoys the privilege and is allowed to move. The move takes its state in a new state which is a function of its own state and the states of its neighbors. Thus after a finite number of moves the predicate becomes true for the global state and the algorithm terminates. Thus the basic structure of any self-stabilizing algorithm can be stated by the pseudo code shown in algorithm 1.

Chapter 3

General Discussion

The distributed systems considered here consist of finite state machines. This report states the basic ideas of the algorithms proposed by different researchers. For some important papers, the correctness proofs are also summarized and the execution steps are shown by diagrams.

The basic ideas of all the papers are almost same. Some papers use a little modified definition of self-stabilization based on system specifications and constraints. During the execution of a self-stabilizing algorithm, if more than one privilege exist in some step, the system takes the help of a daemon to select the privilege. Some algorithms assume the existence of a central daemon while some assume the existence of a distributed daemon.

The requirement of a central demon is an unreasonable constraint for a truly distributed system. In particular, its implementation requires some form of centralized control. If more than one privileges are available, then the central daemon arbitrarily selects one privilege from them.

Dijkstra used such a mechanism because the transitions of neighboring processes were interfering. That is, by firing its enabled transition, one process could disable an enabled transition in another process. In general, a system in which transitions are executed atomically and only sequentially (as provided by a central demon) will not behave the same as one in which transitions are fired in parallel, or one in which reads and writes are interleaved. A distributed daemon is more desirable than the central daemon because the implementation of central daemon is difficult. However, many systems assume the presence of a central daemon for ease of calculation.

If there exists no multi-writer variables and no infinite sequence of moves involving a proper subset of nodes consisting of a pair of adjacent nodes only, then a solution that works with central daemon also works with distributed daemons [23].

The proof methods of almost all self-stabilizing algorithms are also similar. The straightforward way for proving the property of self-stabilization of a system is to first prove that the system can always make a computation step as long as it is not stabilized, and then to find a bounded function whose value monotonically decreases with computing steps. However, some papers like [5, 103] proved correctness without using any bounded function. In [7], correctness is proved using induction method and in [103, **?**] correctness is proved using graph theoretical reasoning.

The complexity analysis of a self-stabilizing algorithm is somewhat complicated. The most of the papers in the literature of self-stabilizing systems did not analyze the time and memory complexities of the proposed algorithms. However, some papers analyzed the complexities either partially or completely.

Chapter 4

Early Research

4.1 Introduction of the Self-stabilization Concept

The idea of self-stabilization was introduced by Dijkstra [35]. In the years following his introduction very few works were done in this area. The most of the works available in the literature are very recent. Dijkstra [35], in his proposal, considered $N + 1$ machines, numbered from 0 through N, in a ring and following notations for machine $nr.i$:

L : the state of its lefthand neighbor, machine $nr.(i-1)mod(N+1)$,

S : the state of itself, machine $nr.i$,

R : the state of its righthand neighbor, machine $nr.(i+1)mod(N+1)$.

Machine $nr.0$ and $nr.N$ are called "the bottom machine" and "the top machine" respectively.

4.1.1 Solution with Four-state Machines

Each machine state is represented by two booleans xS and upS. The values of upS for the bottom machine and the top machine are $true$ and $false$ respectively. The privileges are defined as follows:

for the bottom machine:

$xS = xR$ and non $upR \Rightarrow xS :=$non xS

for the top machine:

$xS \neq xL \Rightarrow xS :=$ non xS

for the other machines:

$xS \neq xL \Rightarrow xS :=$ non xS; $upS := true$

$xS = xR$ and upS and non $upR \Rightarrow upS := false$

4.1.2 Solution with Three-state Machines

Here each machine state is represented by an integer value $S(0 \leq S < 3)$. The states of the bottom machine and the top machine are characterized by B and T respectively. The privileges are defined as follows:

for the bottom machine:

$(B + 1 = R) \Rightarrow B := B + 2$

for the top machine:

$(L = B \wedge T \neq B + 1) \Rightarrow T := B + 1$

for the other machines:

$(L = S + 1) \vee (S + 1 = R) \Rightarrow S := S + 1$

4.1.3 Proof of the Solution with Three-state Machines

Later in [36] Dijkstra provided the proof of his solution for three-state machines. To prove he represented the ring of machines by a string starting with B followed by S's and ending with T. In the string an arrow is placed between neighbors whose states differ such that in the direction of arrow the state decreases (*mod* 3) by 1. The transformations of the corresponding moves are interpreted in terms of arrows.

A variable y is defined as the summation of the number of left-pointing arrows and twice the number of right-pointing arrows.

For Bottom:

From $B \leftarrow R$ to $B \rightarrow R$, $\Delta y = 2 - 1 = +1$

For Top:

From $L \rightarrow T$ to $L \leftarrow T$, $\Delta y = 2 - 1 = +1$

From $L\ T$ to $L \leftarrow T$, $\Delta y = 1 - 0 = +1$

For other machines:

From $L \rightarrow T\ R$ to $L\ S \rightarrow R$, $\Delta y = 2 - 2 = 0$

From $L\ S \leftarrow R$ to $L \leftarrow S\ R$, $\Delta y = 1 - 1 = 0$

From $L \rightarrow S \leftarrow R$ to $L\ S\ R$, $\Delta y = 0 - 3 = -3$

From $L \rightarrow S \rightarrow R$ to $L\ S \leftarrow R$, $\Delta y = 1 - 4 = -3$

From $L \leftarrow S \leftarrow R$ to $L \rightarrow S\ R$, $\Delta y = 2 - 2 = 0$

For proving self-stabilization in the system it is sufficient to prove that within a finite number of moves there is precisely one arrow in the string. Between two successive moves of *Top* at least one move of *Bottom* takes place and a sequence of moves in which *Bottom* does not move is finite. Then it is proved that there is only one arrow in the string after a

finite number of moves. So it can be concluded that the system converges to a legitimate state after a finite number of steps.

4.2 Maintaining the Structure of a Tree

Following the introduction of Dijkstra [35], the work proposed by Kruijer [87] in 1978 can be regarded as another milestone to accelerate the research in the area of self-stabilization. The proposed algorithm maintains the structure of a tree in a distributed system. The algorithm considers the distributed system where machines with an even number(≥ 4) of states are placed in the nodes of the tree. The design is such that in the legitimate states more than one privilege can be present which can logically permit the concurrent operation of the involved machines.

Kruijer's self-stabilizing algorithm considers a tree T with n nodes numbered $1, 2, \cdots n$. The structure of T is characterized by means of an integer array $sup[1:n]$:

$sup[i] = 0$ iff i is the root of T, otherwise $sup[i] = k$ with $k(1 \leq k \leq n)$ being the superior of i. If k is the node next to i on the path from i to the root of T then k is called the superior of i while i is called the subordinate of k.

Each node of T is a $2K$-state($K \geq 2$) machine. The state of each machine i is defined by two variables: an integer variable $s[i]$ with range $0, 1, \cdots K-1$ and a boolean variable $eq[i]$. 0 is considered as an artificial root and it maintains a constant state $s[0] = K$ over all configurations.

Two rules R_0 and R_1 shown in algorithm 2 define the privileges of the machines.

The legitimate states of the system are:

- the so-called perfect states: $s[1] = s[2] = \cdots s[n]$ and $eq[1] = eq[2] = \cdots eq[n] = true$.
- the states that arise from perfect states by the completion of one or more permissible moves.

In the legitimate states only the root can make a move.

The boolean procedure $test(i)$ used in rule R_0 in algorithm 2 is defined as follows:

1. if i is a terminal node of T, $test(i)$ always renders the value $true$;
2. if i is not a terminal node of T and has $k, 1, \cdots, p$ as its subordinates, $test(i)$ renders the value $true$ iff $s[k] = s[1] = \cdots s[p] = s[i]$ and $eq[k] = eq[1] = \cdots = eq[p] = true$ and renders the value $false$ otherwise.

Algorithm 2 Kruijer's tree structure algorithm

```
(R_0)
if (noneq[i] ∧ test(i)) then
   eq[i] = true
end if
(R_1)
if (eq[i] ∧ s[i] ≠ s[sup[i]]) then
   if (sup[i]=0) then
      s[i] := (s[i] + 1)modK
   else
      s[i] := s[sup[i]]
   end if
   eq[i] := false
end if
```

A complete execution of Kruijer's [87] algorithm is shown in figure 4.2 for the tree structure consisting of three nodes of figure 4.1. In the tree structure of figure 4.1, the root is 1, and nodes 2 and 3 are its subordinates. Each node represents a 4-state machine. Hence the values of of the variables $s[i]$(i=1,2,3) are only 0 and 1 and those of the variables $eq[i](i = 1, 2, 3)$ are 0 and 1 (i.e. false and true).

The first configuration in the figure 4.2 shows the initial state which is perfect. The value on the left hand side of each node i shows the value of $s[i]$ and that on the right hand side shows the value of $eq[i]$. The rules by which a node enjoys a privilege are shown in the figure and the underlined rule is the one that is selected for the next move. After five moves the system reaches another perfect state(the last configuration in figure 4.2) where only the root has a privilege.

In this proposed system, for each leaf node(i.e. the nodes at the deepest level l) the procedure *test* delivers the value *true*. For each node at level $l-1$, *test* also delivers *true* value. Moreover, each terminal node can be merged into its superior at level $l-1$ which converts T to a tree with level $l-1$. Hence, in each state of the system at least one privilege is present. Regardless of the initial state and regardless of the privilege selected each time for the next move, Kruijer showed that the system would find itself in a perfect state after a finite number of moves.

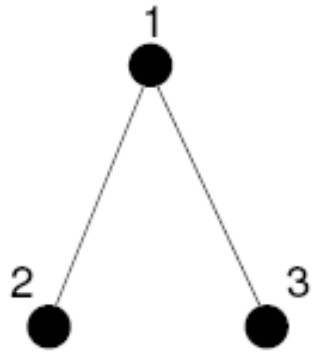


Figure 4.1: Structure of the system

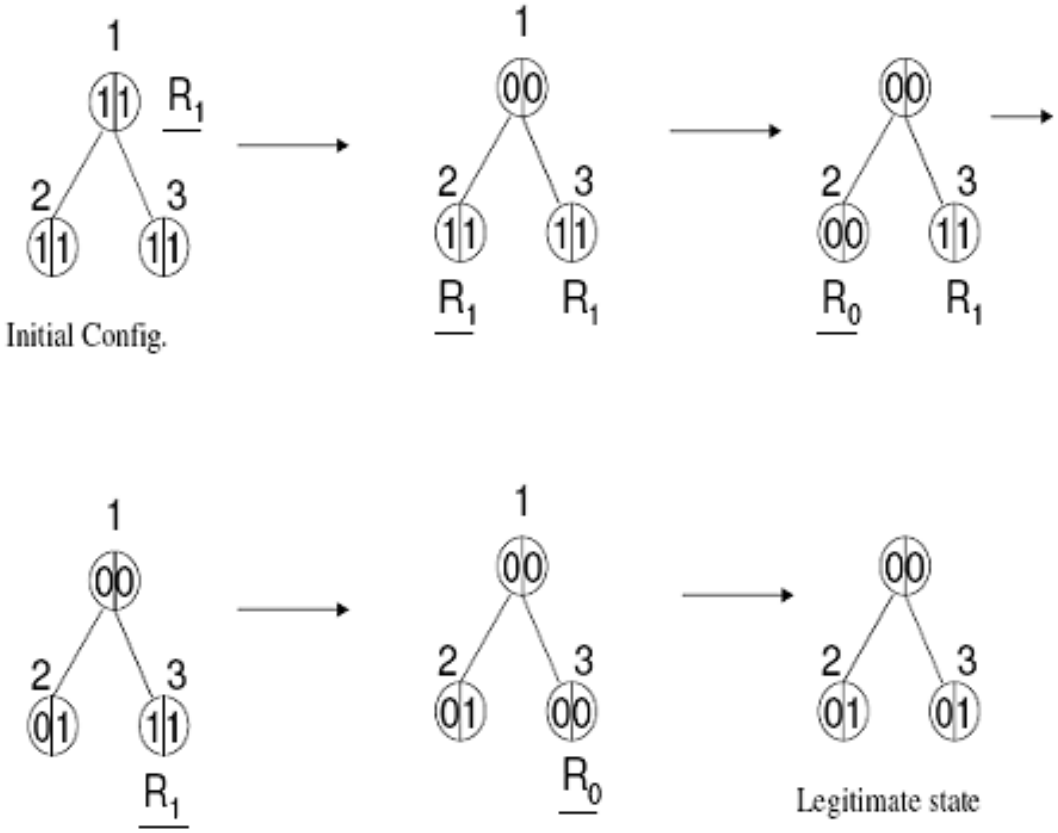


Figure 4.2: Execution of Kruijer's algorithm

4.3 Other Early Works

Herman [68] proposed a probabilistic self-stabilizing algorithm for a unidirectional communication ring with identical processes. The algorithm circulates a single token in the ring. If the initial state of the ring is abnormal, the algorithm executes and the ring converges to a normal state with one token. If the number of processors in the ring is even, the algorithm self-stabilizes to a state without tokens.

Katz and Perry [86], explored the possibility of extending an arbitrary program into a self-stabilizing one. The computational model used by them is that of an asynchronous distributed message-passing system whose communication topology is an arbitrary graph. They contrasted the difficulties of self-stabilization in this model with those of the more common shared-memory models.

Chapter 5

Later Research

5.1 Spanning Tree Construction

5.1.1 Algorithm Proposed by Chen et.al.

In [24] Chen et.al. proposed a self-stabilizing algorithm for constructing spanning trees in distributed systems. The algorithm constructs, from a graph $G = (V, E)$, a spanning tree rooted at a specific node r. Each node i other than the root maintains two variables $L(i)$ and $P(i)$ that represent its level and parent respectively, where $1 \leq L(i) \leq n(|V| = n)$ and $P(i)$ is a neighbor of node i ($P(i)$ is also denoted by p). The initial values of these variables are unpredictable but within their domain. The root node r has a constant level $L(r) = 0$ and no parent variable. In the desired spanning tree $L(r) = 0$ and for any other node $i \neq r$, $L(i) = L(p) + 1$.

The algorithm consists of one predicate and three rules R_0, R_1, R_2 shown in algorithm 3. The predicate is defined as:

$GST \equiv (\forall i, p : i \neq r \wedge p = P(i) : L(i) = L(p) + 1)$

When GST is true the system is in legitimate state.

If the antecedent part of any rule is true then the processor enjoys the privilege and makes the corresponding move. Figure 5.1 shows a full execution of the algorithm. In the figure the parent of a node i is the node which is pointed to by i. Starting from an initial state the system eventually reaches a legitimate state. The rules by which a node enjoys a privilege are shown in the figure and the underlined rule is the one that is selected for the next move. In figure 5.1 the top left configuration indicates the initial state and after six moves the system reaches the final configuration, a spanning tree rooted at node c (the bottom left configuration). At this state no node has the privilege.

Algorithm 3 Spanning tree construction algorithm

(R_0) $L(i) \neq n \wedge L(i) \neq L(p) + 1 \wedge L(p) \neq n$

$\Rightarrow L(i) := L(p) + 1$

(R_1) $L(i) \neq n \wedge L(p) = n \Rightarrow L(i) := n$

(R_2) Let k be some neighbor of i,

$L(i) = n \wedge L(k) < n - 1$

$\Rightarrow L(i) := L(k) + 1; P(i) := k.$

if (GST is true) **then**

the system is in legitimate state

end if

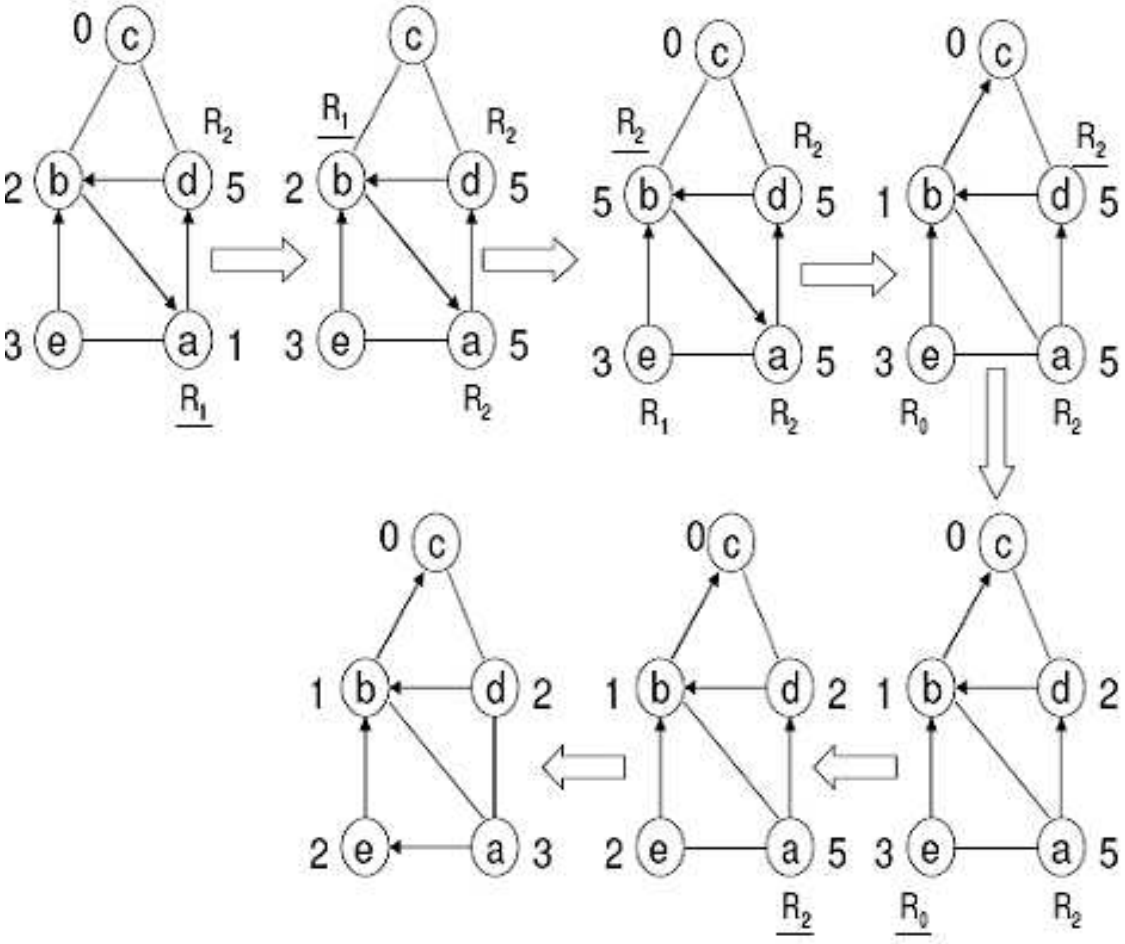


Figure 5.1: Execution of spanning tree construction algorithm

For verification of the algorithm, a parent pointer $i \rightarrow p$ is defined as a *Well-Formed (WF)* pointer if $L(i) \neq n, L(p) \neq n$ and $L(i) = L(p) + 1$. For any configuration, if only the WF pointers are considered, then the nodes of the graph get partitioned. Thus each configuration is, in fact, a spanning forest. For any configuration, each tree is called a WF set denoted by $S_i^{L(i)}$, where i is the root of of the tree. An evaluation function F over the configuration of the system is defined as $(t_0, t_1, \cdots, t_n), 0 \leq t_i < n$ where $t_k (0 \leq k \leq n)$ is the number of WF sets $S_i^{L(i)}$ such that $L(i) = k$. For each configuration of the system represented in figure 5.1 the WF sets and the corresponding F are shown in table 5.1.

step	WF sets	F
0	$S_c^0 = \{c\}, S_a^1 = \{a, b, e\}, S_d^5 = \{d\}$	$F = (1, 1, 0, 0, 0, 1)$
1	$S_c^0 = \{c\}, S_b^2 = \{b, e\}, S_a^5 = \{a\}, S_d^5 = \{d\}$	$F = (1, 0, 1, 0, 0, 2)$
2	$S_c^0 = \{c\}, S_e^3 = \{e\}, S_a^5 = \{a\}, S_b^5 = \{b\}, S_d^5 = \{d\}$	$F = (1, 0, 0, 1, 0, 3)$
3	$S_c^0 = \{c, b\}, S_e^3 = \{e\}, S_a^5 = \{a\}, S_d^5 = \{d\}$	$F = (1, 0, 0, 1, 0, 2)$
4	$S_c^0 = \{c, b, d\}, S_e^3 = \{e\}, S_a^5 = \{a\}$	$F = (1, 0, 0, 1, 0, 1)$
5	$S_c^0 = \{c, b, d, e\}, S_a^5 = \{a\}$	$F = (1, 0, 0, 0, 0, 1)$
5	$S_c^0 = \{c, b, d, e, a\}$	$F = (1, 0, 0, 0, 0, 0)$

Table 5.1: WF sets and values of F of different steps of the algorithm

Before GST is true, there must exist some WF set $S_i^{L(i)}, i \neq r$. For $L(i) \neq n$, node i will enjoy privilege either by rule R_0 or by R_1 and for $L(i) = n$, node i will enjoy privilege by rule R_2. So the algorithm does not terminate until GST is true. The system converges towards the configuration with a single WF set S_r^0 at which $F = (1, 0, \cdots, 0)$. Therefore considering the lexicographic comparison it is verified that F decreases monotonically each time a rule is applied. And after a finite number of moves the system reaches a legitimate state.

5.1.2 Spanning Tree Using DFS

Collin and Dolev [26] proposed an algorithm for constructing a spanning tree using depth first search in a communication graph. The proposed system consists of n processors $P_1, P_2, \cdots, P_n$ where P_1 is defined as the *special* processor(root) and all others are *regular*. Each processor can communicate with its neighbors using shared registers. Any processor, P_i can write in one register, r_i and can read from the register of any of its neighbors. Each

pair of neighbors, P_i and P_j, are connected by an edge $e = (P_i, P_j)$ that supports two-way communication. Each processor P_i orders its edges by some arbitrary ordering α_i. For any edge $e = (P_i, P_j)$, $\alpha_i(j)$ ($\alpha_j(i)$, respectively) denotes the *edge index* of e according to α_i (α_j, respectively). The value of $\alpha_j(i)$ is known to processor P_i. The register of any processor P_i consists of a path field denoted by $path_i$. During the execution of the algorithm the special processor P_1 repeatedly writes the path $\perp$ in $path_1$. All other processors repeatedly read the registers of their neighbors. Any path $path_j$ read by P_i from the neighbor P_j, derives a path for P_i simply by the concatenation: $path_j \circ \alpha_j(i)$. The proposed idea is shown in algorithm 4.

Algorithm 4 DFS algorithm

root P_1 :
while true **do**
  $path_1 := \perp$
end while
non-root P_i :
while true **do**
  for $j := 1$ to δ **do**
    $read\_path_j := read(path_j)$
  end for
  write $path_i$:=min$\{|read_path_j \circ \alpha_j(i)|N$, such that $1 \leq j \leq \delta$ $\}$
end while

Every processor P_i, after reading the stabilized paths of its neighbors, can identify the tree edges and non-tree incident on it. The edge $e = (P_i, P_j)$ is:

(i) an incoming tree edge iff $path_i = path_j \circ \alpha_j(i)$

(ii) an outgoing tree edge iff $path_j = path_i \circ \alpha_i(j)$

(iii) a backward non-tree edge iff $path_j$ is a prefix of $path_i$ and e is not an incoming tree edge.

(iv) a forward non-tree edge iff $path_i$ is a prefix of $path_j$ and e is not an outgoing tree edge. Hence after the execution of algorithm 4 a DFS-spanning tree can be identified.

5.1.3 Breadth First Tree Construction

Huang and Chen [74] gave a self-stabilizing algorithm for constructing breadth-first tree from a connected graph. They used a slightly modified definition of self-stabilization. When the system is in legitimate state their algorithm gets deadlocked in the sense that no computing step can be performed.

In the proposed algorithm a specific node r, for the model graph $G = (V, E)$, is selected as the root. Each node other than r maintains two variables $L(i)(2 \leq L(i) \leq n, n = |V|)$ and $P(i) \in N_i$ which represent the level of i and the parent of i respectively and N_i is the set of neighbors of i. The root node maintains a constant level $L(r) = 1$ and has no parent. The desired breadth-first tree bears the property $(\forall i \neq r)L(i) = (L(p_i) + 1)$ and $L(p_i) = min(\{L(j)|j \in N_i\})$. According to this property, they defined the following predicate to identify the legitimate state of the system:

$BFT \equiv (\forall i : i \neq r : L(i) = L(p_i) + 1) \wedge L(p_i) = min(\{L(j)|j \in N_i\})$

To enjoy the privilege and make the move each node other than the root whose state is never changed has two rules, R_0 and R_1. Algorithm 5 shows the rules.

Algorithm 5 Breadth-first tree construction algorithm

(R_0) $L(i) \neq L(p_i) + 1 \wedge L(p_i) < n$

$\Rightarrow L(i) := L(p_i) + 1$

(R_1) Let k be the neighbor of i such that $L(k) = min(\{L(j)|j \in N_i\})$,

$L(p_i) > L(k) \Rightarrow L(i) := L(k) + 1, p_i := k.$

if (BFT is true) **then**

the system is in legitimate state.

end if

If the antecedent part of any rule is true then the processor enjoys the privilege and makes the corresponding move. For verification of the algorithm, Huang and Chen failed to find an evaluation function for their proposed rules. Then they split the rules R_0, R_1 into another set of rules(M_0, M_1, M_2) that have the equivalent effect.

(M_0) $L(i) \leq L(p_i) < n \Rightarrow L(i) := L(p_i) + 1$

(M_1) $L(i) > L(p_i) + 1 \Rightarrow L(i) := L(p_i) + 1$

(M_2) Let k be a set of neighbor of i such that $L(k) = min(\{L(j)|j \in N_i\})$,

$L(p_i) > L(k) \Rightarrow p_i := k.$

Figure 5.2 shows a full execution of the algorithm considering rules M_0, M_1, M_2. In the figure the parent of a node i is the node which is pointed to by i. The rules by which a node enjoys a privilege are shown in the figure.

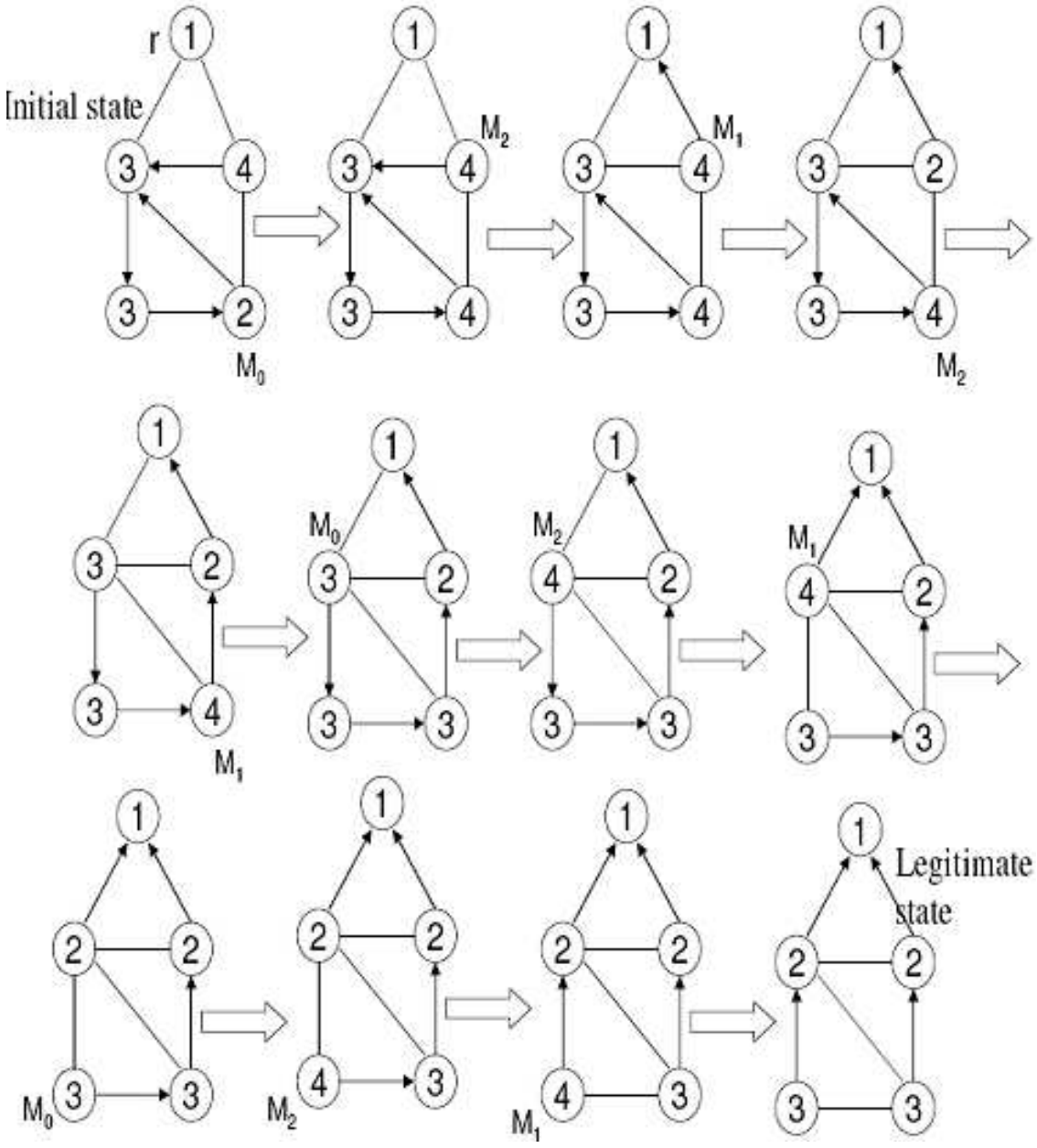


Figure 5.2: Execution of BFT algorithm

Using new rules they defined two functions F_1, F_2 and considered $F \equiv (F_1, F_2)$ as the bounded function. F_1 is defined as

$(t_2, t_3, \cdots, t_n)$

Here t_k is the number of k-turn nodes in the system and for any node i, if $L(i) \leq L(p_i)$ then node i is called a k-turn node, where $k = L(i)$.

And F_2 is defined as

$\sum_{i \neq r}(L(i) + L(p_i))$

For each configuration of the system represented in figure 5.2 the values of functions F_1 and F_2 are shown in table 5.2.

step	F_1	F_2
0	$(1, 1, 0, 0)$	23
1	$(0, 2, 0, 0)$	27
2	$(0, 2, 0, 0)$	25
3	$(0, 2, 0, 0)$	23
4	$(0, 2, 0, 0)$	22
5	$(0, 2, 0, 0)$	20
6	$(0, 1, 0, 0)$	21
7	$(0, 1, 0, 0)$	19
8	$(0, 1, 0, 0)$	17
9	$(0, 0, 0, 0)$	18
10	$(0, 0, 0, 0)$	17
11	$(0, 0, 0, 0)$	16

Table 5.2: Values of F_1 and F_2 at different steps of BFT algorithm

F_1 decreases each time rule M_0 is applied but does not increase if M_1 or M_2 is applied. If node i applies $M - 1$, then L_i decreases and $L(p_i)$ remains unchanged and for any $j, P_j = i, L(j)$ remains unchanged and $L(p_j)$ decreases. When rule M_2 is applied , $L(p_i)$ decreases and $L(i)$ is unchanged for node i. Therefore, F_2 decreases each time rule M_1 or M_2 is applied. Eventually, after a finite number of states the system reaches a legitimate state.

Sur and Srimani [103] gave another idea for constructing BFS spanning tree from a bipartite graph $G = (V, E)$ with $|V| = n$. A specific node r is defined as the root.

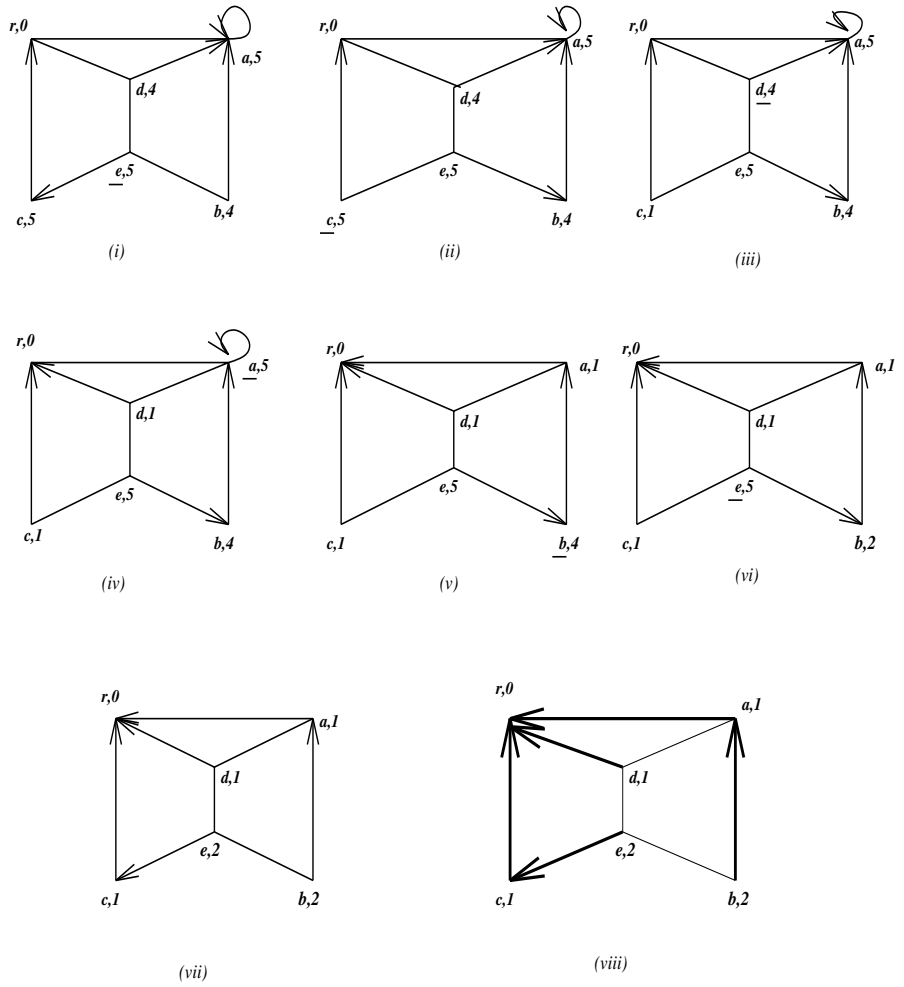


Figure 5.3: Execution of the algorithm of Sur and Srimani

Each node i maintains two variables $L(i)$ and $P(i)$ representing its level and parent respectively. For a node i, $N(i)$ is its set of neighbors and $S(i)$ is the of its neighbors with minimum level. The root has a constant level $L(r) = 0$ for all other nodes $0 \leq L(i) \leq n-1$. Each node enjoys privilege and makes move using a single rule R.

(R): $i \neq r \wedge L(S(i)) \neq n-1 \wedge \{L(i) \neq L(S(i)) + 1 \vee P(i) \notin S(i)\}$

$\Rightarrow L(i) = L(S(i)) + 1; P(i) = k, k \in S(i)$

The system reaches the legitimate state when the following predicated is true:

$\forall i \neq r : L(i) = L(S(i)) + 1 \wedge P(i) \in S(i)$

Figure 5.3 shows a complete execution of the proposed algorithm. In figure 5.3 the top left configuration shows the initial state of the system. The node enjoying the privilege is underlined. After seven moves the system reaches the legitimate state, the bottom right configuration in figure 5.3. The correctness of the algorithm is proved using graph theoretical reasoning which can be applied to prove other self-stabilizing algorithms also.

5.1.4 Minimum-Depth Search of Graphs

The self-stabilizing minimum-depth search(MDS) algorithm proposed by Chaudhuri [23] constructs a spanning tree from a connected undirected graph $G = (V, E)$. The state of each node i is characterized by two variables $d(i)$ and $p(i)$ that represent its depth(level) and parent respectively in tree T rooted at a specific node r in G. The set $N(i)$ represent the neighbor nodes of i in G.

Algorithm 6 MDS algorithm

$(i = r) \wedge (d(r) \neq 0 \vee p(r) \neq r)$

$\Rightarrow d(r) := 0; p(r) := r$

$(i \neq r) \wedge (\sim min\_depth(i) \vee improper\_pr\_info(i))$

$\Rightarrow d(i) := min\forall_{j \in N(i)}\{d(j) + 1\};$

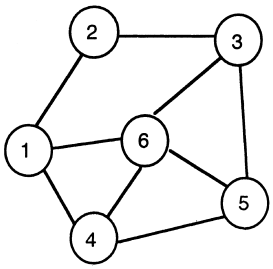
$p(i) := min\forall_{k \in N(i)}\{k | d(k) = min\forall_{j \in N(i)}\{d(j) + 1\}\}$

In a legitimate state the following invariants hold:

1. For the root r, $(d(r) = 0) \wedge (p(r) = r)$;
2. For all other nodes, $p(i) \in N(i) \wedge \sim \exists_{j \in N(i) - \{p(i)\}}\{d(j) < d(i) - 1\}$.

Two predicates for each node $i \neq r$ are defined as follows

1. $min\_depth(i) : d(i) = min\forall_{j \in N(i)}\{d(j) + 1\}$



(a)

node	1	2	3	4	5	6
depth	5	4	1	3	6	2
parent	3	5	4	6	1	4
privilege	**	*	*		*	*

(b)

node	1	2	3	4	5	6
depth	3	4	1	3	6	2
parent	6	5	4	6	1	4
privilege		*	*		**	*

(c)

node	1	2	3	4	5	6
depth	3	4	1	3	2	2
parent	6	5	4	6	3	4
privilege		**	*			*

(d)

node	1	2	3	4	5	6
depth	3	0	1	3	2	2
parent	6	2	4	6	3	4
privilege	*		**			*

(e)

node	1	2	3	4	5	6
depth	3	0	1	3	2	2
parent	6	2	2	6	3	4
privilege	**					*

(f)

node	1	2	3	4	5	6
depth	1	0	1	3	2	2
parent	2	2	2	6	3	4
privilege				*		**

(g)

node	1	2	3	4	5	6
depth	1	0	1	3	2	2
parent	2	2	2	6	3	1
privilege				**		

(h)

node	1	2	3	4	5	6
depth	1	0	1	2	2	2
parent	2	2	2	1	3	1
privilege						

(i)

Figure 5.4: Execution of MDS algorithm

2. $improper\_pr\_info(i) : p(i) \notin N(i) \vee d(i) \neq d(p(i)) + 1$

The root may be privileged by perturbation. Once it makes a move it never becomes privileged again. Any node other than the root may be privileged due to a move made by one of its adjacent nodes. The rules are shown in algorithm 6.

The algorithm is illustrated with the help of an example in figure 5.4. Figure 5.4(a) shows a given arbitrary connected undirected graph with $n = 6$ and $r = 2$. Figure 5.4(b) shows an arbitrary initial (illegitimate) state of the system. A single asterisk ($*$) in the privilege row for various nodes indicates that the corresponding node enjoys the privilege, whereas a double asterisk ($**$) indicates that the corresponding node is selected to make a move. A possible sequence of moves made by the algorithm during its execution are shown in subsequent configurations. Figure 5.4(i) shows the legitimate state. The correctness of the algorithm is proved using a simple reasoning based method.

5.1.5 Arbitrary Spanning Tree Construction

An algorithm was proposed by Antonoiu and Srimani [5] for constructing arbitrary spanning tree from a connected graph $G = (V, E)$ with $|V| = n$. A specific node r is defined as the root. Each node i maintains two variables $L(i)(0 \leq L(i) \leq n)$ and $P(i)(0 \leq P(i) \leq n)$ representing its level and predecessor pointer respectively. For a node i, $N(i)$ is its set of neighbors. For each node i two predicates, Ψ_i and Φ_i are defined as follows:

$\Psi_i = ((P(i) \in N(i)) \wedge (L(P(i)) + 1))$

$\Phi_i = (\exists j)(j \in N(i) \wedge L(j) < L(i))$

Algorithm 7 Arbitrary spanning tree construction algorithm

if $(i = r \wedge (P(i) \neq r \vee (L(i) \neq 0)))$ **then**
  $P(i) = r; L(i) = 0;$
else
  if $(\sim \Psi_i \wedge (L(i) < n) \wedge \sim \Phi_i)$ **then**
    $L(i) = L(i) + 1$
  else
    if $(\sim \Psi_i \wedge \Phi_i)$ **then**
      $P(i) = j; L(i) = L(j) + 1$
    end if
  end if
end if

Algorithm 7 explains the single rule for a node i that defines the privilege. The system is in legitimate state when $((L(r) = 0) \wedge (P(r) = r) \wedge (\forall i \neq r : \Psi_i))$

Figure 5.5 illustrates the execution of the algorithm from an arbitrary initial state (figure(a)) of a connected graph with 6 nodes where r is the root. Each node in the figure is labelled with its name and its level. The predecessor pointer at each node is shown by a dotted line. The set PV denotes the set of privileged nodes and the set AV denotes the set of active nodes. The configuration in figure(d) is in the legitimate state. The correctness of the algorithm is proved without using any bounded function.

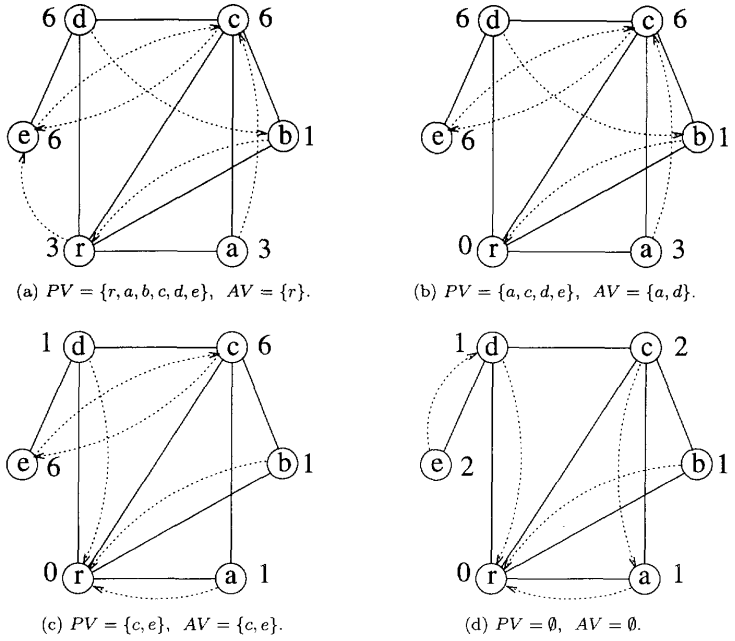


Figure 5.5: Execution of arbitrary spanning tree construction algorithm

5.1.6 Other Spanning Tree Constructing Algorithms

Garg and Agarwal [49] proposed a self-stabilizing algorithm based on the idea of core and non-core states for maintaining a spanning tree in a completely connected graph. It provides a method for changing the root of the tree dynamically. Here Neville's third encoding is used to compute a labeled tree. The algorithm stabilizes faster than other previous approaches.

A self-stabilizing algorithm for minimum spanning tree computation in an arbitrary undirected graph is proposed in [7]. The algorithm consists of a uniform rule for each

node of the graph. Each node i maintains two arrays $D_i[1..n]$ and $L_i[1..n]$. The value of $D_i[j]$ for all $i, j \in V$, at any system state gives the cost of minimum α-cost path from i to j. The value of $L_i[j]$ denotes the level of i with respect to the implicit tree rooted at j. In the legitimate state, each node knows which of its incident edges belong to the minimum spanning tree of the graph. The correctness is proved using induction method. Aggarwal and Kutten [2] presented a time-optimal self-stabilizing algorithms for distributed spanning tree computation in asynchronous networks. They presented both a randomized algorithm for anonymous networks as well as a deterministic version for ID-based networks. Antonoiu and Srimani [3] proposed a self-stabilizing for minimal spanning tree in a symmetric graph. The algorithm proposed in [10] can construct spanning trees in wireless ad hoc networks.

5.2 Finding Maximal Matching in Distributed Networks

Hsu and Huang [73] proposed a self-stabilizing algorithm for finding maximal matching in distributed networks. The model is represented by the graph $G = (V, E)$, where $|V| = n$. Each node i knows its neighbors $N(i)$ and maintains a pointer represented by $i \rightarrow j$ when i selects $j \in N(i)$ to match. $i \rightarrow null$ means that i does not select anyone to match.

If $i \rightarrow j$, then i's state denoted by $S.i$ can be of three types:

1. $S.i = waiting$ if $(i \rightarrow j) \wedge (j \rightarrow null)$;
2. $S.i = matching$ if $i \Leftrightarrow j$ i.e. $i \rightarrow j \wedge j \rightarrow i$
3. $S.i = chaining$ if $i \rightarrow j \wedge j \rightarrow k \wedge k \neq i$

If $i \rightarrow null$, then two possible states are:

1. $S.i = dead$ if $(i \rightarrow null) \wedge (\forall j : j \in N(i) : S.j = matched)$;
2. $S.i = free$ if $(i \rightarrow null) \wedge (\exists j : j \in N(i) : S.j \neq matched)$;

In the legitimate state, the system can find the maximal matching i.e. each node's state must be either *matched* or *dead*.

Therefore, when the following predicate is true, the system reaches a legitimate state.
$GMM \equiv (\forall i :: S.i = matched \vee S.i = dead)$

Three rules R_1, R_2, R_3 define the privileges. The rules are explained in algorithm 8

Algorithm 8 Hsu's algorithm for finding maximal matching

$(R_1)(i \rightarrow null) \wedge (\exists j : j \in N(i) : j \rightarrow i)$

$\Rightarrow i \rightarrow j$

$(R_2)\ (i \rightarrow null) \wedge (\forall k : k \in N(i) :\sim (k \rightarrow i))$

$(\exists j : j \in N(i) : j \rightarrow null)$

$\Rightarrow i \rightarrow j$

$(R_3)\ i \rightarrow j \wedge j \rightarrow k \wedge k \neq i$

$\Rightarrow i \rightarrow null$

if (GMM is $true$) **then**

the system is in legitimate state.

end if

If GMM is false, then there must exist some node i whose state is neither matched nor dead. If $S.i$ is chaining, then R_3 can be applied, if $S.i$ is waiting, then R_1 can be applied, if $S.i$ is free, then R_1 or R_2 or R_3 can be applied based on the state of $j \in N(i)$. Therefore, if GMM is false, at least one node can enjoy privilege. If m, d, w, f, and c denote the total number of nodes whose state are matched, dead, waiting, free and chaining respectively, then a bounded function $F \equiv (m + d, w, f, c)$ is defined whose value increases with each move and converges to $(n, 0, 0, 0)$ in the legitimate state. Hence the algorithm terminates after a finite number of moves.

5.3 Finding Shortest Path in a Distributed System

Huang [75] encoded the self-stabilizing algorithm for finding the shortest path using two rules R_0 and R_1 shown in algorithm 9. The shortest path between the nodes i and j is denoted by $d(i, j)$. $N(i)$ is the set of neighbors of node i. Each node i maintains a local variable $d(i)$ whose value is in the range $\{0, 1, 2, \cdots\}$. A specific node r is selected as the source node. When the system represented by the graph $G = (V, E)$ reaches a legitimate state, then $\forall i \in V, d(i) = d(i, r)$.

For ease of proof rule R_1 is split into two rules.

$(R_l(a))\ d(i) < min_{j \in N(i)}(d(j) + w(i, j))) \Rightarrow d(i) := min_{j \in N(i)}(d(j) + w(i, j)))$

$(R_1(b))\ d(i) > min_{j \in N(i)}(d(j) + w(i, j))) \Rightarrow d(i) := min_{j \in N(i)}(d(j) + w(i, j)))$

A node $i \neq r$ is called a turn node whenever $d(i) < min_{j \in N(i)}(d(j) + w(i, j))$ and is called a k-turn node if i is a turn node and $d(i) = k$. $A^{(k)}$ is the set of all k-turn nodes in the system and $t^k = |A^{(k)}|$ is the cardinality of $A^{(k)}$. An evaluation function F is defined

Algorithm 9 Huang's algorithm for finding shortest path

For source r

(R_0) $d(r) \neq 0 \Rightarrow d(r) := 0$

For node $i \neq r$

(R_1) $d(i) \neq min_{j \in N(i)}(d(j) + w(i,j)) \Rightarrow d(i) := min_{j \in N(i)}(d(j) + w(i,j))$

when the $Predicate \equiv (d(r) = 0) \wedge (\forall i \neq r, d(i) = min_{j \in N(i)}(d(j) + w(i,j)))$ is true the system is in legitimate state.

as $F \equiv (F_1, F_2)$. $d_{init}(i)$ is the $d(i)$ in the initial state and the value $d_u(i)$ is defined as

1. $d_u(r) = d_{init}(r)$;
2. for $i \neq r$, $d_u(i) = max\{d_{init}(i), d_u(p) + w(i,p)\}$, where p is the parent of i in tree T rooted at r.

Then F_1 is defined as $(t_0, t_1, \cdots, t_m), m = max_{i \in V} d_u(i)$ and F_2 is defined as $\sum_{i \in V} d(i)$. During the execution of the algorithm F_1 decreases each time rule $R_1(a)$ is applied and F_2 decreases each time rule R_0 or $R_1(b)$ is applied. And thus F converges to $(0, 0, \cdots, 0)$. Hence the system stabilizes after a finite number of steps.

5.4 Finding Articulation points, Bridge, and Biconnected Components

The algorithm proposed by Karaata [81] for finding cut-nodes uses the spanning tree constructed from breadth first search of the graph. The algorithm is based on the idea that a vertex v is an articulation point if and only if there exists two neighbors of node v in the spanning tree that are not transitively linked. Paths P_i and P_j are referred to as link paths of non-tree edge (i,j). $e(i)$ or e-set denotes a set of non-tree edges on node i and its descendants. $e(v) = \{(i,j)\}$ indicates that node v is on the link paths P_i and P_j. $e(v) = \{\{(i,j)\}\}$ indicates that the node v is the lowest common ancestor of nodes i and j. The following predicates are defined:

non_tree(i): denotes whether or not non-tree edge x is incident on node i and edge x is not in $e(i)$.

lca(i): denotes whether or not node i has exactly two children that contain edge x in their e-sets and $\{x\}$ is not in $e(i)$.

single_child(i): denotes whether or not node i has exactly one child that contains non-tree edge x in its e-set and edge x is not in $e(i)$.

no_child(i):denotes whether or not node i does not have any child that contains edge x in its e-set and edge x is not incident on i and edge x is in $e(i)$.

not_lca(i): denotes whether or not node i is not the lowest common ancestor of nodes p and q, however, $e(i)$ contains $\{x\}$, where $x = (p, q)$.

Algorithm 10 depicts the rules for moves.

Algorithm 10 Algorithm for finding articulation points

$non\_tree(i) \vee single\_child(i) \Rightarrow e(i) = e(i) \cup \{x\}$

$lca(i) \Rightarrow e(i) = e(i) \cup \{\{x\}\}$

$no\_child(i) \Rightarrow e(i) = e(i) - \{x\}$

$not\_lca(i) \Rightarrow e(i) = e(i) - \{\{x\}\}$

When the algorithm terminates, based on the e-sets of neighbors, each node i can determine whether it is a cut node or not. If node i has two neighbors u, w incident on tree edges such that $e(u)$ and $e(w)$ are not transitively linked with respect to the set of e-sets of neighbors of i incident on tree edges, then i is a cut node. The author gave the correctness proof of his proposed method.

In figure 5.6 a graph with a BFS tree rooted at node 1 is shown where each tree edge is shown by solid lines and each non-tree edge is shown by dashed lines. Path $P_2 = 2, 1$ and $P_3 = 3, 1$ are the link paths of non-tree edge $(2, 3)$, and link paths $P_5 = 5, 4, 3$ and $P_6 = 6, 3$ are link paths of non-tree edge $5, 6$. $e(3)$ contains $\{(5, 6)\}$ and $e(1)$ contains $\{1, 2\}$. $e(1)$ and $e(6)$ have no common element and therefore 1 and 6 are not transitively linked. Hence, node 3 is a cut-node.

Using the same idea as in [81], Karaata and Chaudhury [84] proposed another algorithm for finding bridges of a graph. The computation steps, complexities of the algorithms proposed in [81] and [84] are same.

Devismes [34] proposed another algorithm for finding cut-nodes and bridges which is faster than that of Karaata. In [82], Karaata proposed another self-stabilizing algorithm based on the algorithms of [81] and [84] to find the biconnected components of a connected undirected graph. The algorithm uses the spanning tree constructed by BFS in the graph. The algorithm bases on the idea that two fundamental cycles belong to the same biconnected component if and only if they are transitively connected. He also provided the complete proof of his algorithm.

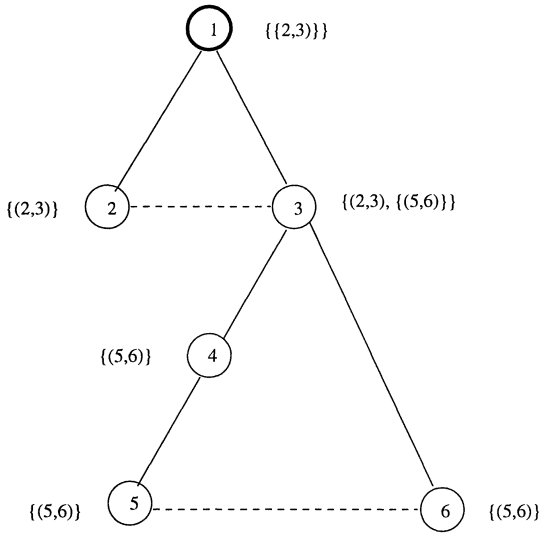


Figure 5.6: State of a system after termination of articulation point finding algorithm

5.5 Graph Coloring

Ghosh and Karaata [58] proposes a coloring algorithm on a directed acyclic version of a given planar graph and a self-stabilizing algorithm for generating the directed acyclic version of the planar graph. The authors also provide the combined algorithm. The algorithm works with no more than six colors.

A set of colors $K = \{0, 1, 2, \cdots, D-1\}$ and a directed acyclic graph in which the maximum out degree of each node is $D-1$ are considered in the first phase. $C[i]$ denotes the color of node i. $succ(i)$ denotes the set of nodes each of which is connected with an outgoing edge from node i and $succolor[i]$ represents the set of colors of all the nodes in $succ[i]$.

For DAG generation, the edge directions of the graph are adjusted in such a manner that, eventually no node has an out degree greater than five. $setofx[i]$ represents the set of x-values of the nodes in $succ[i]$.

The combination of two algorithms is shown in algorithm 11.

Gradinariu and Tixeuil [63] propose three self-stabilizing solutions for coloring the vertices of an arbitrary graph. Two solutions are deterministic and one is randomized. The solutions are based on a greedy technique. These can be used to solve directed acyclic orientation as well as maximal independent set with no additional cost. Shukla [102] proposed al algorithm for coloring chains and oriented rings via systematic randomization.

Algorithm 11 Algorithm for coloring planar graphs

for node i

$\exists j : j \in succ[i]$ ::

$(outdegree[i] \leq 5) \wedge (C[i] = C[j]) \wedge (b \in (K - succolor[i]))$

$\Rightarrow C[i] := b$

$outdegree[i] > 5$

$\Rightarrow x[i] := (max\ setofx[i])+1$

Hedetniemi et.al. [66] proposed a much faster algorithm for proper coloring of an arbitrary system graph.

5.6 Finding Cycles, Centers, Medians in a Graph

Chaudhury [22] proposed an algorithm for detecting the fundamental cycles in a graph. The algorithm uses the spanning tree T rooted at r which is constructed by DFS from a graph $G = (V, E)$. The following notations are used:

$n(i)$: sets of neighbors of node i in G

$p(i)$: parent node of i in $T(r)$ $(p(r) = \emptyset)$ $c(i)$: set of children of i in $T(r)$

$nt(i)$: set of non-tree edges incident on i

$C(i, j)$: fundamental cycles created by the non-tree edge (i, j)

$a(i)$: set of ancestors of $i \in V$ $d(i)$: set of descendants of $i \in V$

$s(i)$: set of all non-tree edge ids such that each of these edges connects a descendant of i to a proper ancestor of i

$su(i)$: $\bigcup_{\forall j \in c(i)} s(j)$

$fc(i)$: set of all non-tree edge ids such that the fundamental cycles created by each of these edges passes through i.

The algorithm is based on the idea that every non-tree edge uniquely defines a fundamental cycle of G when it is added to $T(r)$. In the legitimate state, the following invariants hold:

1. For a leaf node $i(c(i) = \emptyset) : s(i) = nt(i); fc(i) = nt(i)$

2. For a non-leaf node $i(c(i) = \emptyset) : s(i) = nt(i) \cup su(i) - nt(i) \cap su(i); fc(i) = su(i)$

The rules for defining privilege are shown in algorithm 12. In the legitimate state, for each node $i \in V$, each edge id in $fc(i)$ defines a unique fundamental cycle.

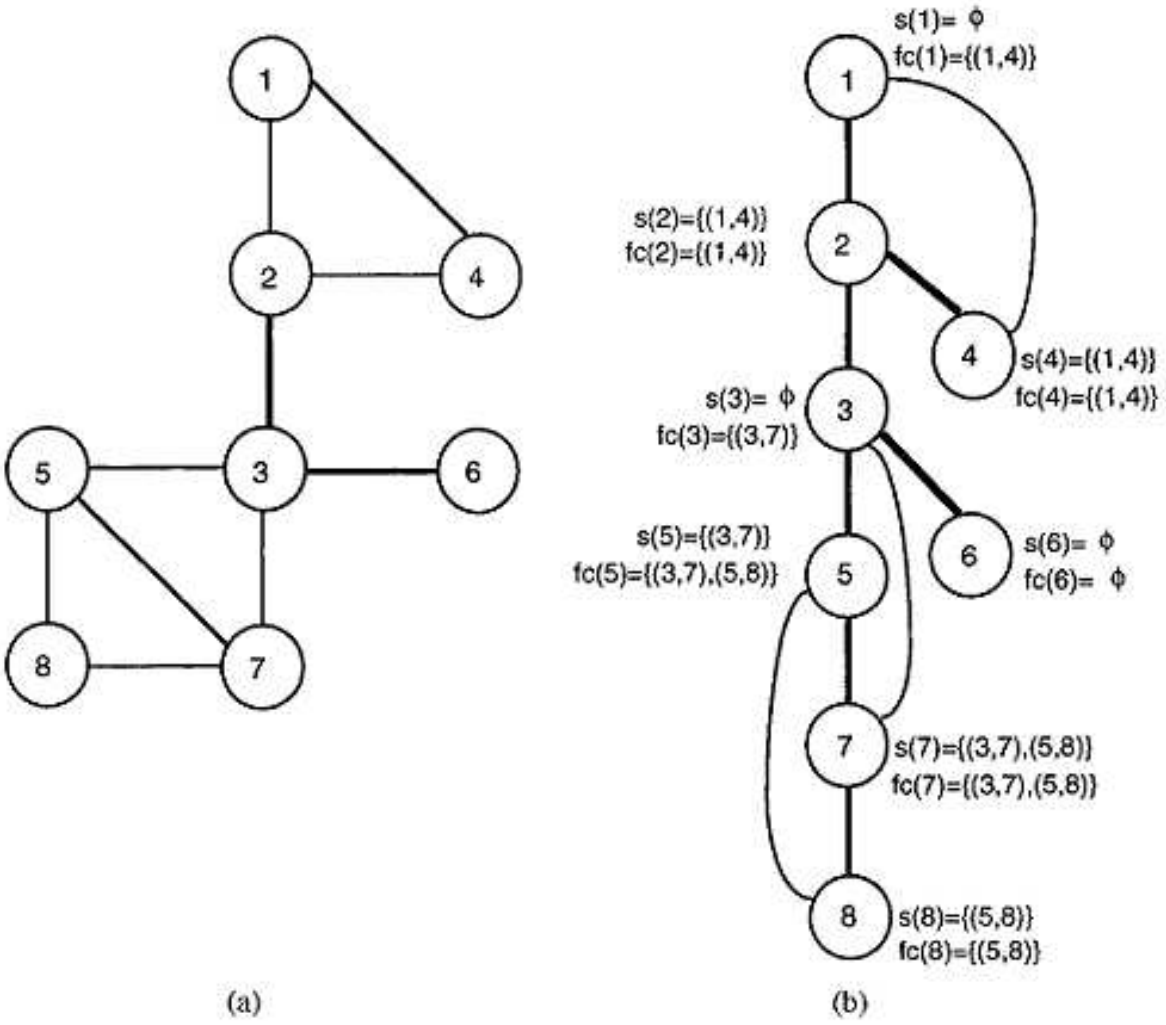


Figure 5.7: Execution of cycle detecting algorithm

Algorithm 12 Algorithm for detecting cycles

$i(c(i) = \emptyset) \wedge (s(i) \neq nt(i) \vee fc(i) \neq nt(i))$

$\Rightarrow s(i) := nt(i); fc(i) := nt(i)$

$i(c(i) \neq \emptyset) \wedge (s(i) \neq nt(i) \cup su(i) - nt(i) \cap su(i) \vee fc(i) \neq su(i))$

$\Rightarrow s(i) := nt(i) \cup su(i) - nt(i) \cap su(i); fc(i) := su(i)$

Figure 5.7 shows the output of the algorithm(on the right) on an undirected graph G(shown on the left). The bridges of G are shown by bold lines. The figure on the right side shows a DFS tree rooted at node 1. The tree edges are shown by bold lines. The final values of $s(i)$ and $fc(i) \forall i \in V$ are also shown.

Karaata and Pemmaraju [85] presented a self-stabilizing algorithm to detect the centers and medians of trees. For each vertex two values $h - value$ and $s - value$ are defined. To motivate the algorithm two conditions for these two values are given. A central scheduler arbitrarily selects an enabled guard and allows the execution of the corresponding atomic move to be completed before any guard is re-evaluated. When all guards are false, the system reaches a state where the values satisfy their conditions. At this state, the vertex with maximum $h - value$ is the center and that with maximum $s - value$ is the median.

Antonoiu and Srimani [8] proposed another self-stabilizing algorithm for finding the median of a tree graph. Each node needs to maintain an ordered list of its neighbors. The algorithm is dominated by a single rule for every node. A leaf node can enjoy privilege only once and an internal node may become privileged again after one of its neighbors takes an action. When the system reaches the legitimate state, no node can be privileged. The correctness of the algorithm is proved by mathematical induction in an interesting way that can be used to prove other self-stabilizing algorithms also. Self-stabilizing approach was also be used to find the 2-centers of a tree [76].

5.7 Finding Maximal 2-Packing

The self-stabilizing algorithm for finding maximal 2-packing was proposed by Gairing et. al. [47]. The algorithm has six rules to define the privileges for the nodes. The rules are shown in algorithm 13.

Each node i in the network has a unique identifier $id(i)$, and there exists a total ordering of these identifiers. Each node i also maintains a boolean variable $x(i)$ whose value is true if i is an element in the 2-packing the algorithm tries to construct, and false otherwise. Moreover, each node has a pointer that can point to any neighbor $j \in N(i)$ or to null represented by $i \rightarrow j$ or $i \rightarrow null$ respectively.

Algorithm 13 Gairing's Algorithm for finding maximal 2-packing

R_1 :
if $x(i) = 0 \wedge i \rightarrow null \wedge$
$\forall j \in N(i) : x(j) = 0 \wedge (j \rightarrow i \vee j \rightarrow null)$ **then**
$x(i) = 1$;
end if
R_2 :
if $x(i) = 0 \wedge i \rightarrow j \wedge$
$\forall k \in N(i) : x(k) = 0$ **then**
$i \rightarrow null$;
if $\forall l \in N(i) : l \rightarrow i \vee l \rightarrow null$ **then**
$x(i) = 1$
end if
end if
R_3 :
if $x(i) = 0 \wedge (i \rightarrow null \vee (i \rightarrow j \wedge x(j) = 0))$
$\exists k \in N(i) : x(k) = 1$ **then**
$i \rightarrow k$, where $id(k) = min\{id(l) : l \in N(i) \wedge x(l) = 1\}$
end if
R_4 :
if $x(i) = 1 \wedge \exists j \in N(i) : x(j) = 1$ **then**
$x(i) = 0$
$i \rightarrow j$, where $id(j) = min\{id(l) : l \in N(i) \wedge x(l) = 1\}$
end if
R_5 :
if $x(i) = 1 \wedge \forall j \in N(i) : x(j) = 0 \wedge \exists k \in N(i) : k \rightarrow l, l \neq i$ **then**
$x(i) = 0$;
$i \rightarrow null$
end if
R_6 :
if $x(i) = 1 \wedge \sim (i \rightarrow null)$
$\forall j \in N(i) : x(j) = 0 \wedge (j \rightarrow i \vee j \rightarrow null)$ **then**
$i \rightarrow null$
end if

5.8 Self-Stabilizing Mutual Exclusion

The idea of self-stabilization was used for designing mutual exclusion protocols for networks by many researchers. Beauquier and Delat [12] gave a probabilistic self-stabilizing algorithm for mutual exclusion in uniform rings. The paper [14] also focuses on self-stabilizing mutual exclusion and leader election. Buskens et.al [19] gave another self-stabilizing mutual exclusion protocol in the presence of faulty nodes.

Dolev et.al. [37] proposed a mutual exclusion protocol for tree structured systems, a spanning tree protocol for any connected graph, and a third protocol by use of fair protocol combination. The result protocol is a self-stabilizing mutual exclusion protocol for dynamic systems. It is based on the assumption that read or write operations are atomic for the shared memory. Antonoiu and Srimani [4] proposed a protocol for mutual exclusion between neighboring nodes. This protocol also stabilizes using read/write atomicity. They also proposed a leader election protocol for tree graphs in [6]. The paper [13] gives an analysis for memory requirements for self-stabilizing leader election protocols.

The mutual exclusion algorithm proposed in [30] uses an unfair distributed scheduler while that in [31] uses an arbitrary scheduler. Kakugawa [79] gave an algorithm for mutual exclusion on unidirectional rings under distributed daemon. Yen [109] proposed a mutual exclusion algorithm which was highly safe. Nesterenko and Mizuno [94] proposed a quorum-based self-stabilizing distributed mutual exclusion algorithm. A technique for verifying mutual exclusion algorithms is proposed in [99].

5.9 Other Works

The algorithm proposed by Kakugawa and Ishii [78] consists of four guarded commands. Each process p in the network G is given a set of its neighbor processes as input, and finds a set of its neighbors that are fully connected together with p. The algorithm detects the cliques in the graph once the system is in legitimate state.

Two nodes i, j of a graph belong to the same strongly connected component if and only if there exists a path from i to j and vice versa. Based on this idea, Karaata and Al-Anzi [83] proposed a self-stabilizing algorithm to find the strongly connected components of a directed graph. In [21] Chaudhury presented a self-stabilizing algorithm for finding bridge-connected components of a graph in $O(n^2)$ time.

The notion of self-stabilization is also used in flow networks. A self-stabilizing distributed algorithm was proposed in [56] for finding the maximum flow in a flow network.

Each node in the flow network G except the source node contains a process that asynchronously makes moves based on local information only. Each move updates the local state of the corresponding process.

In [70] it is shown that fault containment, within a single step, is probabilistically achievable for many stabilizing programs without implying replication overhead. The paper also introduces a transformation procedure to convert a stabilizing program into a fault-containing stabilizing program.

In [18] a general self-stabilizing scheme is given for solving any synchronization problem whose safety specification can be defined using a local property. The proposed solution significantly improves all the existing self-stabilizing approaches which are quadratic in the number of states. The paper also proposes an approach to transform any serial system to a distributed system.

Ghosh and Bejan [52] examined two different safety models-strong and weak models for stabilizing distributed systems and analyzed the cost of enforcing safety requirements pertaining to different failures. The paper considers contamination number, maximum number of processes that can change state before the system reaches a legal state, as an important criteria for safety. The framework provided in this paper for enforcement of safety in stabilizing systems help formalize the problem of safe stabilization and accommodate different kinds of failures that may have implications on safety but not on stabilization.

Awerbauch et.al. [9] introduced self-stabilizing end-to-end communication protocol in fail-stop networks and reset protocol in dynamic networks. The generalized self-stabilizing end-to-end communication protocol depends on the capacity channel and the size of the messages. To make the reset protocol self-stabilizing, it is made locally checkable and then all link predicates necessary to ensure correct operation are listed. And finally, local correction of links are specified.

Ghosh et. al. [54] introduces the notion of fault containment in distributed self-stabilizing systems. They give a framework for specifying and evaluating fault-containing self-stabilizing protocols. They also present a transformer to map any non-reactive self-stabilizing algorithm into an equivalent fault-containing self-stabilizing algorithm.

Petit and Villain [97] proposed a self-stabilizing depth-first token circulation protocol for uniform rooted networks. They explained how the basic depth-first token circulation protocol is nearly self-stabilizing and how to obtain a self-stabilizing protocol by just adding what is necessary to destroy cycles. The proposed algorithm is very convenient to obtain the mutual exclusion or to construct a spanning tree. The depth-first token

circulation protocol proposed in [32] works in an arbitrary rooted network.

The basic problem of persistent bit, where the system is required to maintain a value in the face of transient failures by means of replication is considered in [88]. It proposes an algorithm to recover the value quickly. The algorithm can recover the value of the bit at all nodes in $O(f)$ time, where f is a transient fault hit.Moreover, complete state quiescence occurs in $O(d)$ time units, where d denotes the diameter of the network. The paper also gives a transformation procedure to convert a distributed non-reactive and non-stabilizing protocol into a self-stabilizing one.

In [72], Hsu and Huang presented an approach to analyze the self-stabilizing algorithms with the finite state machine model. From the rules of the self-stabilizing algorithm, this approach defines some states and derives a state transition diagram. They used the self-stabilizing maximal matching algorithm [73] as an example to illustrate how the approach works. In [53] a simple self-stabilizing leader election algorithm is proposed for an oriented ring with bidirectional communication capabilities. During the execution of the algorithm only the faulty node and its neighbors change their states to converge to a stable state. The system stabilizes in constant time from a single transient fault. Blair and Manne [16] proposed a new set of tree rooting algorithms. The algorithm has one rule for the first phase. When the system stabilizes in the first phase, each node can determine the number of nodes in the entire network. In the second phase, rooting a tree is done using four rules.

Flatebo and Datta [44] showed that it was possible to design self-stabilizing algorithms requiring only two states and in [45] they proposed two-state algorithms for rings. In [1], Abello and Dolev showed that any computable problem could be realized in a self-stabilizing fashion. They derived the result by presenting a distributed system which tolerated transient faults and simulated the execution of a Turing machine.

Self-stabilizing approaches are also being used for argumentation. The proposed approach in [11] for argumentation introduces a remarkable flexibility in the management of argumentation activity with respect to a centralized approach.

The idea of self-stabilization is also used for network decomposition. The algorithm proposed in [15] deals with the partitioning problem of a network. The proposed algorithm can adapt in the dynamic system.

Boldi and Vigna [17] proved the existence of an algorithm which allows to stabilize a distributed system to a desired behaviour. Previous proposals required drastic increases in asymmetry and knowledge in order to work, while this algorithm does not use any additional knowledge.

Collin et.al. [25] showed the theoretical bounds on the capabilities of the connectionist architecture and other distributed approaches to constraint satisfaction problem.

The recent papers [27, 28, 29, 32] have some good ideas about self-stabilization in networks. Ghosh et.al. [55] proposed self-stabilizing dynamic programming algorithm on trees. The papers [40, 41, 39, 43, 42] also contain some recent works on self-stabilization. Goddard [61] proposed an algorithm for strong matching in a system graph. A synchronous self-stabilizing minimal domination protocol is offered in [106] for an arbitrary network graph. Some recent analysis and proposals for self-stabilizing systems are available in [46, 48, 50, 51, 57, 59, 60, 62, 77, 90, 89, 101, 105, 104].

Lin et.al. [91] proposed an efficient algorithm for finding a maximal independent set. In [80] Karaata proposed a dynamic self-stabilizing algorithm for constructing transport net. Recently, 3-edge-connectivity has been studied in the context of self-stabilization [110, 111, 112].

Chapter 6

Current Research

The area of self-stabilizing systems is being proliferated day by day. At present there are many active researchers all over the world as shown in appendix B. Also there are conferences, workshops devoted entirely or partially to this area (Appendix C). To get an up-to-date information about the present works I contacted some active researchers(appendix D). At present they are working on:

1. Self-Stabilizing Microprocessor, Analyzing and Overcoming Soft-Errors.
2. Towards Self-Stabilizing Operating Systems.
3. Self-Stabilizing Distributed File Systems.
4. Self-Stabilizing Autonomic Recovorer for Eventual Byzantine Software.
5. Robust Active SuperTier Systems.
6. Stability of Long-lived Consensus.
7. Stability of Multi-Valued Continuous Consensus.
8. elf-Stabilizing Group Communication in Directed Networks.
9. Polygonal Broadcast, Secret Maturity and the Firing Sensors.
10. HyperTree for Self-Stabilizing Peer-to-Peer Systems.
11. Random Walk for Self-Stabilizing Group Communication in Ad-Hoc Networks.
12. Self-stabilizing location management.
13. Self-stabilizing protocols for sensor networks.

Chapter 7

Review of Research

Analysis of self-stabilizing algorithms is somewhat complicated. The most of the papers did not include the complexity analysis. Therefore, it is hard to compare the algorithms. However, some papers gave the analysis of the proposed algorithms.

Cansell et.al. [20] gave a formal method for analyzing self-stabilizing protocols using predicate diagrams. In [74], Huang and Chen could not explain the complexity of their algorithm. But one referee pointed out that if precedence was assigned for the rules of the proposed algorithm then some redundant moves in constructing a breadth-first tree would be reduced. But Huang and Chen [74] still claims that reduction is not guaranteed for their algorithm, although reduction, in general, is expected.

In [24], Chen did not explain the complexity of his algorithm. In self-stabilizing algorithm non-interfering property is desirable, but the algorithm proposed by Chen [24] had interfering rules. Yet the rules work properly even if they are not atomic.

In the legitimate states in the self-stabilizing system proposed by Kruijer [87] more than one privilege can be present which can logically permit the concurrent operation of the involved machines. This sounds somewhat strange but this would make it useful to choose an implementation of the system which makes concurrent operation of the machines physically possible. The legitimate state defined in [87] is also somewhat different. In every legitimate state a root can make a move. Kruijer [87] also proved an important property of his proposed system. If the system is in a legitimate state and the common value of the variables $s[i](i = 1, 2, \cdots n)$ is s_0, then the system will again reach another legitimate state after exactly $2n$ moves and in this new legitimate state the common value of the variables $s[i](i = 1, 2, \cdots n)$ is $(s_0 + 1) \bmod K$. There is no complexity analysis in his paper.

The idea proposed by Collin et. al. [26] that I have explained in subsection 5.1.2 can

be used for other graph algorithms using different order relations. The space complexity and time complexity of the algorithm are $O(nlog\Delta)$ and $O(dn\Delta)$ respectively, where Δ is an upper bound on the degree of a node and d is the diameter of the graph.

It is often very hard to find an evaluation function for proving the stabilizing property of the system. For some cases this can be overcome by transforming the set of rules into another set of rules with same eventual effect. In [74, 75] Huang adopted this concept.

Hsu [73] analyzed the complexity of his algorithm. As explained in section 5.2 the value of $m+d+w+f+c$ in F is always equal to n, and the values of $m+d, w, f$ and c are always between 0 and n individually. In the worst case F can have the value $(0, 0, 0, n)$. The number of steps needed to transfer the value of F from $(0, 0, 0, n)$ to $(n, 0, 0, 0)$ is almost equal to the number of non-negative integer solutions (x_1, x_2, x_3, x_4) for the the equation $x_1 + x_2 + x_3 + x_4 = n$ which is $\frac{(n+1)(n+2)(n+3)}{6}$ assuming $x_1 = m + d, x_2 = w, x_3 = f, x_4 = c$. Hence in the worst case, the upper bound $O(n^3)$ is obtained. The average time complexity was not analyzed.

The MDS algorithm of Chuadhuri [23] that I discussed in subsection 5.1.4 assumes the existence of a central daemon, yet it can be easily established that the proposed algorithm also works with distributed daemon. The author analyzed the complexity of his algorithm. The algorithm makes at most $O(n^2)$ moves.

Karaata [81] gave the complexity analysis for the algorithm of finding articulation point. The algorithm takes $O(n^2|E)$ moves. The algorithm for finding bridges also takes the same time [84]. But the algorithm proposed in [34] takes $O(n^2)$ time. The algorithm for finding bi-connected components [82] terminates after $O(d)$ time, where d is the diameter of the biconnected component with the largest diameter in the graph. The algorithm for detecting strongly connected components proposed in [83] takes $O(C)$ rounds to compute strongly connected components, where C is the length of the longest cycle in the graph.

The graph coloring algorithm of [58] does not guarantee coloring the nodes with less than six colors but the idea of this paper can be used for coloring non-planar graphs also. In [63] the system stabilizes within $O(n \times B)$ time, where B is the degree of the graph. The solutions can be used to solve directed acyclic orientation as well as maximal independent set with no additional cost.

The algorithm for detecting cycles in graph Chaudhury [22] takes $O(n^2)$ time if the algorithm uses an already constructed DFS spanning tree, otherwise it will take $O(n^3)$ time. The clique finding algorithm in [78] converges in $O(n^4)$ steps, where n is the number of processors. The algorithm in [56] finds the maximum flow in $O(n^2)$ moves.

Blair and Manne [16] analyzed the complexity of their efficient self-stabilizing algorithms for tree networks. It takes $O(n^2)$ moves. The two-state algorithms for token rings in [45] stabilizes in $O(n^2)$ time, where n is the number of machines in the network.

Although, all the proposals I explained in the previous chapters considered machines with finite state in the distributed system, Dolev et.al. [37] considered machines with infinite state also. Yen [109] also considered infinite-state machines in the system.

Chapter 8

Conclusion and Future Works

The area of self-stabilization is so important, and interesting as well, that within the past few years there has been a flurry of papers, as well as some workshops devoted entirely to this area. This report emphasizes some papers which are regarded as milestones and also summarizes the ideas and views of almost all other papers currently available in the literature of self-stabilizing systems.

There are many promising future works in this area. Hsu [73] observed two problems in his maximal matching finding algorithm. Reducing the upper bound time complexity and relaxing the requirement of R_1 and R_2 from the case of testing the pointer of node i and the pointers of all its neighbors to the case of simply testing its own pointer and only one neighbor's pointer are left for future works. Chen et.al. [24] left the interfering issue of their algorithm for investigation in future.

Combining the idea of [26] and self-stabilizing leader election protocols, a self-stabilizing DFS algorithm for a system of processors with distinct identifiers can be designed. In [81] finding the tight complexity bound considering the complexity of BFS is an open problem. The algorithm takes $O(n^2|E)$ moves and it is said that it is optimal. Proving that an optimal self-stabilizing algorithm takes $O(n^2|E)$ moves is also an open problem. A tight performance analysis of the algorithm in [78] is left as a future work. Finding triconnected components, three-edge connected components, separation-pairs are also promising future works.

Bibliography

[1] Abello, J., and Dolev, S. On the computational power of self-stabilizing systems. *Theoretical Computer Science 182*, 1–2 (1997), 159–170.

[2] Aggarwal, S., and Kutten, S. Time optimal self-stabilizing spanning tree algorithm. In *Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science(FSTTCS93), Springer-Verlag LNCS:761* (1993), pp. 400–410.

[3] Antonoiu, G., and Srimani. A self-stabilizing distributed algorithm for minimal spanning tree in a symmetric graph. *Computers and Mathematics with Applications 35*, 10 (May 1998), 15–23.

[4] Antonoiu, G., and Srimani, P. Mutual exclusion between neighboring nodes in a tree that stabilizes using read/write atomicity. In *European Conference on Parallel Processing* (September 1998), pp. 545–553.

[5] Antonoiu, G., and Srimani, P. K. A self-stabilizing distributed algorithm to construct an arbitrary spanning tree of a connected graph. *Computers and Mathematics with Applications 30*, 9 (November 1995), 1–7.
Keywords: Self-stabilizing algorithms, distributed daemon, spanning tree, correctness proof, analysis.
This paper proposes a self-stabilizing approach to maintain an arbitrary spanning tree in a connected graph. Each node i maintains two data structures $L(i)$ and $P(i)$ representing level and predecessor pointer respectively. Whenever the system is in illegitimate state at least one of the nodes should be able to recognize it and should take some correct action. The algorithm consists of a single uniform rule for all the nodes in the graph. In the legitimate state the system presents a valid rooted spanning tree of the graph.

[6] Antonoiu, G., and Srimani, P. K. A self-stabilizing leader election algorithm for tree graphs. *Journal of Parallel and Distributed Computing 34*, 2 (May 1996), 227–232.

[7] Antonoiu, G., and srimani, P. K. Distributed self-stabilizing algorithm for minimum spanning tree construction. *Presented in European Conference on Parallel processing* (1997), 480–487.

A self-stabilizing algorithm for minimum spanning tree computation in an arbitrary undirected graph is proposed in this paper. The algorithm consists of a uniform rule for each node of the graph. The system reaches a legitimate state in finite number of steps. At this stage, each node knows which of its incident edges belong to the minimum spanning tree of the graph. The correctness is proved using induction method.

[8] Antonoiu, G., and Srimani, P. K. A self-stabilizing distributed algorithm to find the median of a tree graph. *Journal of Computer and System Science 58*, 1 (February 1999), 215–221.

This paper proposes a self-stabilizing algorithm for finding the median of a tree graph. Each node needs to maintain an ordered list of its neighbors. The algorithm is dominated by a single rule for every node. A node satisfying the rule is said to have privilege. A node cannot enjoy privilege after its move. A leaf node can enjoy privilege only once and an internal node may become privileged again after one of its neighbors takes an action. When the system reaches the legitimate state, no node can be privileged. The correctness of the algorithm is proved by mathematical induction in an interesting way that can be used to prove other self-stabilizing algorithms also.

[9] Awerbuch, B., Patt-Shamir, B., and Varghese, G. Self-stabilization by local checking and correction (extended abstract). In *IEEE Symposium on Foundations of Computer Science* (1991), pp. 268–277.

This paper introduces self-stabilizing end-to-end communication protocol in fail-stop networks and reset protocol in dynamic networks. A simple method for local checking and correction is used. The generalized self-stabilizing end-to-end communication protocol depends on the capacity channel and the size of the messages. To make the reset protocol self-stabilizing, it is made locally checkable and then all link predicates necessary to ensure correct operation are listed. And finally, local correction of links are specified.

[10] Baala, H., Flauzac, O., Gaber, J., Bui, M., and El-Ghazawi, T. A. A self-stabilizing distributed algorithm for spanning tree construction in wireless ad hoc networks. *Journal of Parallel and Distributed Computing 63*, 1 (January 2003), 97–104.

[11] Baroni, P., and Giacomin, M. A distributed self-stabilizing algorithm for argumentation. In *Proceedings 15th International Parallel and Distributed Processing Symposium* (23-27 April 2001), IEEE, p. 8 pp.

[12] Beauquier, J., and Delat, S. Probabilistic self-stabilizing mutual exclusion in uniform rings. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing, Los Angeles, California, United States* (1994), SIGACT: ACM Special Interest Group on Algorithms and Computation Theory SIGOPS: ACM Special Interest Group on Operating Systems, ACM Press New York, NY, USA, p. 378.

[13] Beauquier, J., Gradinariu, M., and Johnen, C. Memory space requirements for self-stabilizing leader election protocols. In *Symposium on Principles of Distributed Computing* (1999), pp. 199–207.

[14] Beauquier, J., Gradinariu, M., and Johnen, C. Token-based self-stabilizing uniform algorithms. *Journal of Parallel and Distributed Computing 62*, 5 (May 2002), 899–921.

[15] Belkouch, F., Bui, M., and Cheng, L. Self-stabilizing deterministic network decomposition. *Journal of Parallel and Distributed Computing 62*, 4 (April 2002), 696–714.

[16] Blair, J. R. S., and Manne, F. Efficient self-stabilizing algorithms for tree networks. In *Proceedings. 23rd International Conference on Distributed Computing Systems* (19-22 May 2003), IEEE, pp. 20–26.
In this paper a new set of tree rooting algorithms is explained. The algorithm assumes the existence of a static unique identifier for each node. The algorithm has one rule for the first phase. When the system stabilizes in the first phase, each node can determine the number of nodes in the entire network. In the second phase, rooting a tree is done using four rules. The main idea of this paper is the time-honored concept of using a little additional storage that dramatically reduces the computational time.

[17] Boldi, P., and Vigna, S. Self-stabilizing universal algorithms. In *Proceedings of the Third Workshop on Self-Stabilizing Systems* (1997), Carleton University Press, pp. 141–156.

[18] Boulinier, C., Petit, F., and Villain, V. When graph theory helps self-stabilization. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing, St. John's, Newfoundland, Canada* (2004), SIGOPS: ACM Special Interest Group on Operating Systems SIGACT: ACM Special Interest Group on Algorithms and Computation Theory, ACM Press New York, NY, USA, pp. 150–160.

In this paper a general self-stabilizing scheme for solving any synchronization problem whose safety specification can be defined using a local property. The asychronous phase clock, the local mutual exclusion, the local group mutual exclusion problem etc. can be solved using this algorithm. The proposed solution significantly improves all the existing self-stabilizing approaches which are quadratic in the number of states. The paper also proposes an approach to transform any serial system to a distributed system.

[19] Buskens, R. W., and Bianchini, Jr., R. P. Self-stabilizing mutual exclusion in the presence of faulty nodes. In *FTCS-25: 25th International Symposium on Fault Tolerant Computing Digest of Papers* (Pasadena, California, 1995), pp. 144–153.

[20] Cansell, D., Méry, D., and Merz, S. Formal analysis of a self-stabilizing algorithm using predicate diagrams. In *Workshop Integrating Diagrammatic and Formal Specification Techniques (GI-/ÖCG-Jahrestagung)* (Vienna, Austria, 2001), M. Wirsing, Ed., vol. 157/I of *books@ocg.at*, pp. 39–45.

[21] Chaudhuri, P. An o(n^2) self-stabilizing algorithm for computing bridge-connected components. *Computing 62*, 1 (February 1999), 55–67.

[22] Chaudhuri, P. A self-stabilizing algorithm for detecting fundamental cycles in a graph. *Journal of Computer and System Science 59*, 1 (August 1999), 84–93.

The self-stabilizing algorithm proposed in this paper can detect the fundamental cycles of a connected undirected graph in $O(n^2)$ time. The algorithm is applied on the spanning tree constructed by DFS in the graph. The leaf nodes enjoy privilege by perturbation and after move a leaf node cannot enjoy privilege again. Excepting leaf nodes any other node may be privileged after a move made by any of its chil-

dren. Once the system is in legitimate state, each node knows exactly how many fundamental cycles pass through it.

[23] Chaudhuri, P. A self-stabilizing algorithm for minimum-depth search of graphs. *Information Processing Letters 118*, 1-4 (September 1999), 241–249.
Keywords: Self-stabilization, transient faults, undirected graph, minimum depth search.
A minimum-depth search of a connected undirected graph on an asynchronous distributed system is proposed in this paper. The algorithm does not need initialization and is not sensitive to transient faults. It produces a minimum-depth spanning tree of a graph in a self-stabilizing fashion using only $O(n^2)$ moves. The correctness is proved using a simple reasoning-based method.

[24] Chen, N.-S., Yu, H.-P., and Huang, S.-T. A self-stabilizing algorithm for constructing spanning trees [1]. *Information Processing Letters 39*, 3 (August 1991), 147–151.
Keywords: Analysis of Algorithms, combinatorial problems, design of algorithms, fault tolerance, spanning tree, self-stabilizing.
This paper proposes a self-stabilizing algorithm to maintain a spanning tree in distributed system. A connected graph $G = (V, E)$ is used to model the system. A specific node r is selected as the root and the algorithm constructs from G a spanning tree rooted at r. Each node i other than the root maintains two local variables: level of i and parent of i. A predicate is defined whose true value indicates that the system is in legitimate state. The algorithm consists of three rules. A processor enjoys privilege if any of the three rules is satisfied. For any configuration of the system, the nodes of the graph are partitioned into Well-Formed(WF) sets defined by WF pointers. Over any WF set, there is a directed spanning tree. The algorithm runs until the value of the predicate becomes true i.e. the system is in legitimate state. And the number of moves to reach this state is finite.

[25] Collin, Z., Dechter, R., and Katz, S. Self-stabilizing distributed constraint satisfaction. *Chicago Journal of Theoretical Computer Science* (1999).

[26] Collin, Z., and Dolev, S. Self-stabilizing depth-first search. *Information Processing Letters 49*, 6 (March 1994), 297–301.
Keywords: distributed computing, fault tolerance.

[1] Milestone paper

This paper presents a self-stabilizing algorithm for constructing a depth-first tree of a communication graph. The algorith, once started, converges to a consistent global state by itself. The algorithm uses a lexicographic order relation on the path representation. The space complexity and time complexity of the algorithm are $O(nlog\Delta)$ and $O(dn\Delta)$ respectively, where Δ is an upper bound on the degree of a node and d is the diameter of the graph.

[27] Costello, A. M., and Varghese, G. Self-stabilization by window washing. In *Symposium on Principles of Distributed Computing* (1996), pp. 35–44.

[28] Das, S. K., Datta, A. K., and Tixeuil, S. Self-stabilizing algorithms in dag structured networks. *Parallel Processing Letters 9*, 4 (December 1999), 563–574.

[29] Datta, A. K., Gradinariu, M., Kenitzki, A. B., and Tixeuil, S. Self-stabilizing wormhole routing on ring networks. *Journal of Information Science and Engineering 19*, 3 (May 2003), 401–414.

[30] Datta, A. K., Gradinariu, M., and Tixeuil, S. Self-stabilizing mutual exclusion using unfair distributed scheduler. In Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS'00), pp. 465–470.

[31] Datta, A. K., Gradinariu, M., and Tixeuil, S. Self-stabilizing mutual exclusion under arbitrary scheduler. *The Computer Journal 47*, 3 (May 2004), 289–298.

[32] Datta, A. K., Gurumurthy, S., Petit, F., and Villain, V. Self-stabilizing network orientation algorithms in arbitrary rooted networks. In *International Conference on Distributed Computing Systems* (2000), pp. 576–583.

[33] Datta, A. K., Johnen, C., Petit, F., and Villain, V. Self-stabilizing depth-first token circulation in arbitrary rooted network. In *5th International Colloquium on Structural Information and Communication Complexity (SIROCCO'98)* (1998), pp. 32–46.

[34] Devismes, S. A silent self-stabilizing algorithm for finding cut-nodes and bridges. Tech. Rep. 2733, LARIA, CNRS FRE, January 2005.

[35] Dijkstra, E. W. Self-stabilizing systems in spite of distributed control[2]. *Communications of the ACM 17*, 1 (November 1974), 643–644.

[2]Milestone paper

Keywords: multiprocessing, networks, self-stabilization, synchronization, mutual exclusion, robustness, sharing, error recovery, distributed control, harmonious cooperation, self-repair.

This is the first paper to propose the idea of self-stabilization in distributed system. Here a connected graph with a finite state machine in each node is considered. For each machine, one or more privileges are defined. To enjoy the privilege a machine needs to have a truth value of a predefined boolean function of its own state and the states of its neighbours. The machine that experiences the privilege is brought/moved into a new state that is a function of its old state and the states of its neighbors. Regardless of the initial state and regardless of the privilege selected the system is guaranteed to find itself in a legitimate state after a finite number of moves. The paper gives a solution for k-state($k > N$) machines placed in a ring and numbered from 0 to N. The solutions for four-state machines and three-state machines are also given.

[36] Dijkstra, E. W. A belated proof of self-stabilization[3]. *Distributed Computing 1*, 1 (January 1986), 5–6.

This paper provides the correctness proof for the solution with three-state machines that was given in [35]. The ring of these machines is represented by a string starting with B followed by S's and ending with T. The variables B, T, and S represent the bottom, top and normal machines respectively. In the string an arrow is placed between neighbors whose states differ such that in the direction of arrow the state decreases (*mod* 3) by 1. The transformations of the corresponding moves are interpreted in terms of arrows. The paper concludes that for the demonstration of self-stabilization in the system it is sufficient to prove that within a finite number of moves there is precisely one arrow in the string. Between two successive moves of Top at least one move of Bottom takes place and a sequence of moves in which Bottom does not move is finite. Then it is proved that there is only one arrow in the string after a finite number of moves.

[37] Dolev, S., Israeli, A., and Moran, S. Self-stabilization of dynamic systems assuming only read/write atomicity. In *Proceedings of the ninth annual ACM symposium on Principles of distributed computing* (1990), SIGOPS: ACM Special Interest Group on Operating Systems and SIGACT: ACM Special Interest Group on Algorithms and Computation Theory, ACM Press New York, NY, USA, pp. 103–117.

[3]Milestone paper

In this paper three self-stabilizing protocols for distributed systems in the shared memory model are presented. The first one is a mutual exclusion protocol for tree structured systems, the second one is a spanning tree protocol for any connected graph, the third one is obtained by use of fair protocol combination. The result protocol is a self-stabilizing mutual exclusion protocol for dynamic systems. It is based on the assumption that read or write operations are atomic for the shared memory.

[38] Dolev, S., and Schiller, E. Self-stabilizing group communication in directed networks. *Acta Informatica 40*, 9 (September 2004), 609–636.

[39] Ducourthial, B., and Tixeuil, S. Self-stabilizing global computations with r-operators. In *Proceedings of the 2nd International Conference On Principles Of DIStributed Computing (OPODIS'98), Hermes, 1998.* (1998), pp. 99–113.

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

[41] Ducourthial, B., and Tixeuil, S. Self-stabilization with path algebra. *Theoretical Computer Science 293*, 1 (February 2003), 219–236.

[42] Flatebo, M., and Datta, A. K. Self-stabilizing deadlock detection algorithms. In *Proceedings of the 1992 ACM annual conference on Communications, Kansas City, Missouri, United States* (1992), ACM: Association for Computing Machinery, ACM Press New York, NY, USA, pp. 117–122.

[43] Flatebo, M., and Datta, A. K. Simulation of self-stabilizing algorithms in distributed systems. In *Proceedings of the 25th annual symposium on Simulation* (May 1992), vol. 19, SIGSIM: ACM Special Interest Group on Simulation and Modeling, IEEE Computer Society Press Los Alamitos, CA, USA, pp. 32–41.

[44] flatebo, M., and Datta, A. K. Two-state self-stabilizing algorithms. In *Proceedings., Sixth International Parallel Processing Symposium* (March 1992), pp. 198–203.

Keywords: Binary-state machines, distributed algorithms, self-stabilization.

Dijkstra [35] gave self-stabilizing concept for machines with three states or four states. This paper proves that it is possible to design algorithms requiring only two states. All the three algorithms proposed here assume the presence of a central daemon. For algorithm 1, the system stabilizes when the exceptional machine 0 has a

privilege. For algorithm 2, the system stabilizes when the central daemon chooses among the privileged machines randomly. For the third algorithm, the system stabilizes in the presence of a randomized central daemon.

[45] FLATEBO, M., AND DATTA, A. K. Two-state self-stabilizing algorithms for token rings. *IEEE Transactions on Software Engineering 20*, 6 (June 1994), 500–504.
Keywords: Binary-state machines, distributed algorithms, mutual exclusion, self-stabilization.
Although it is shown that a minimum of three states are required by any self-stabilizing algorithm in a ring, this paper gives an algorithm that works for two-state machines in an asynchronous unidirectional ring. The two algorithms other than the first one require randomization. The second algorithm builds on the first one and reduces the number of network connections required. The third algorithm again reduces necessary connections and yields two-state. The system stabilizes in $O(n^2)$ time, where n is the number of machines in the network.

[46] FRIBOURG, L., MESSIKA, S., AND PICARONNY, C. Coupling and self-stabilization. In *Distributed algorithms* (Oct 2004), R. Guerraoui, Ed., vol. 3274/2004 of *Lecture Nodes in Computer Science*, pp. 201–214.

[47] GAIRING, M., GEIST, R. M., HEDETNIEMI, S., AND KRISTIANSEN, P. Self-stabilizing algorithm for maximal 2-packing. *Nordic Journal of Computing 11*, 1 (March 2004), 1–11.
Keywords: Self-stabilizing algorithms, 2-packing, Markov Analysis.
An ID-based self-stabilizing algorithm for finding maximal 2-packing in an arbitrary graph is proposed in this paper. Each node has a unique identifier and each node i maintains a boolean variable $x(i)$ indicating its membership in the desired 2-packing. The paper also shows how to use Markov analysis to analyze the behavior of a non-ID based version of the algorithm on small graphs.

[48] GAIRING, M., HEDETNIEMI, S. T., KRISTIANSEN, P., AND MCRAE, A. A. Self-stabilizing algorithms for k-domination. In *Sixed Symposium on Self-Stabilization (SSS 2003)* (2003), Springer LNCS 2704, pp. 49–60.

[49] GARG, V. K., AND AGARWAL, A. Self-stabilizing spanning tree algorithm with a new design methodology. (available via ftp or www at maple.ece.utexas.edu as technical report tr-pds-2004-001). Tech. Rep. TR-PDS-2004-001, 2004.
Keywords: fault tolerance, self-stabilization, spanning tree.

In this paper a self-stabilizing algorithm is proposed for maintaining a spanning tree in a completely connected graph. It is based on the idea of core and non-core states. It provides a method for changing the root of the tree dynamically. Here Neville's third encoding is used to compute a labeled tree. The algorithm stabilizes faster than other previous approaches.

[50] Gartner, F. C., and Pagnia, H., Eds. *Time-Efficient Self-Stabilizing Algorithms through Hierarchical Structures, 6th International Symposium, SSS 2003, San Francisco, CA, USA, June 24-25, 2003, Proceedings* (2003), vol. 2704 of *Lecture Notes in Computer Science*, Springer.

[51] Ghosh, S. An alternative solution to a problem on self-stabilization. *ACM Transactions on Programming Languages and Systems 15*, 4 (September 1993), 735–742.

[52] Ghosh, S., and Bejan, A. A framework of safe stabilization. In *Self-Stabilizing Systems* (2003), pp. 129–140.
In this paper two different safety models-strong and weak models for stabilizing distributed systems are examined and the cost of enforcing safety requirements pertaining to different failures are analyzed. The paper considers contamination number, maximum number of processes that can change state before the system reaches a legal state, as an important criteria for safety. The framework provided in this paper for enforcement of safety in stabilizing systems help formalize the problem of safe stabilization and accommodate different kinds of failures that may have implications on safety but not on stabilization.

[53] Ghosh, S., and Gupta, A. An exercise on fault-containment: Self-stabilizing leader election. *Information Processing Letters 59*, 5 (September 1996), 281–288.
In this paper a simple self-stabilizing leader election algorithm is proposed for an oriented ring with bidirectional communication capabilities. During the execution of the algorithm only the faulty node and its neighbors change their states to converge to a stable state. The system stabilizes in constant time from a single transient fault.

[54] Ghosh, S., Gupta, A., Herman, T., and Pemmaraju, S. V. Fault-containing self-stabilizing algorithms. In *Symposium on Principles of Distributed Computing* (1996), pp. 45–54.
This paper introduces the notion of fault containment in distributed self-stabilizing systems. It gives a framework for specifying and evaluating fault-containing self-

stabilizing protocols. It also presents a transformer to map any non-reactive self-stabilizing algorithm into an equivalent fault-containing self-stabilizing algorithm.

[55] Ghosh, S., Gupta, A., and Pemmaraju, M. H. K. S. V. Self-stabilizing dynamic programming algorithms on trees. In *Proceedings of the Second Workshop on Self-Stabilizing Systems* (1995), pp. 11.1–11.15.

[56] Ghosh, S., Gupta, A., and Pemmaraju, S. V. A self-stabilizing algorithm for the maximum flow problem. *Distributed Computing 10*, 4 (July 1997), 167–180.
Keywords: Distributed algorithms, fault tolerance, self-stabilization, maximum flow.
This paper proposes a self-stabilizing distributed algorithm for finding the maximum flow in a flow network. The algorithm uses local checking and local correction. Each node in G except the source node contains a process that asynchronously makes moves based on local information only. Each move updates the local state of the corresponding process. The algorithm finds the maximum flow in $O(n^2)$ moves.

[57] Ghosh, S., and He, X. Scalable self-stabilization. *Journal of Parallel and Distributed Computing 62*, 5 (May 2002), 945–960.

[58] Ghosh, S., and Karaata, M. H. A self-stabilizing algorithm for coloring planar graphs. *Distributed Computing 7*, 1 (1993), 55–59.
Keywords: Self-stabilization, distributed algorithm, graph coloring, dag, atomicity.
In this paper a self-stabilizing approach is used to color the nodes of a planar graph with no more than six colors. In the first phase of the algorithm coloring is done on a directed acyclic graph and in the second phase the directed acyclic version of the planar graph is generated by self-stabilizing scheme. The idea of this paper can be used for coloring non-planar graphs also.

[59] Ghosh, S., and Pemmaraju, S. V. Trade-offs in fault-containing self-stabilization. In *Symposium on Principles of Distributed Computing* (1997), p. 289.

[60] Godard, E. A self-stabilizing enumeration algorithm. *Information Processing Letters 82*, 6 (June 2002), 299–305.

[61] Goddard, W., Hedetniemi, S. T., Jacobs, D. P., and Srimani, P. K. Self-stabilizing distributed algorithm for strong matching in a system graph. In *Proceedings of High Performance Computing (HiPC 2003) - 10th International Conference* (Hyderabad, India, 17-20 december 2003), LNCS 2913, Springer Verlag, pp. 66–73.

[62] Gopal, A. S., and Perry, K. J. Unifying self-stabilization and fault-tolerance. In *Proceedings of the twelfth annual ACM symposium on Principles of distributed computing , Ithaca, New York, United States* (1993), SIGOPS: ACM Special Interest Group on Operating Systems SIGACT: ACM Special Interest Group on Algorithms and Computation Theory, ACM Press New York, NY, USA, pp. 195–206.

[63] Gradinariu, M., and Tixeuil, S. Self-stabilizing vertex coloring of arbitrary graphs. *Paper presented at International Conference on Principles of Distributed Systems (OPODIS'2000) in Paris, France* (2000).
In this paper two self-stabilizing deterministic and one self-stabilizing randomized solutions are presented for coloring the vertices of an arbitrary graph in spite of unfair scheduling based on a greedy technique. The system stabilizes within $O(n \times B)$ time, where B is the degree of the graph. The solutions can be used to solve directed acyclic orientation as well as maximal independent set with no additional cost.

[64] He, X. *Controlled Recovery in Self-stabilizing Systems.* PhD thesis, Graduate College, University of Iowa, Iowa, Proquest Information and Learning, 300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA, December 2001.

[65] Hedetniemi, S. M., Hedetniemi, S. T., Jacobs, D. P., and Srimani, P. K. Self-stabilizing algorithms for minimal dominating sets and maximal independent sets. *Computers and Mathematics with Applications 46*, 5-6 (September 2003), 805–811.

[66] Hedetniemi, S. T., Jacobs, D. P., and Srimani, P. K. Linear time self-stabilizing colorings. *Information Processing Letters 87*, 5 (September 2003), 251–255.

[67] Herlihy, M., and Tirthapura, S. Self stabilizing distributed queuing. *Lecture Notes in Computer Science 2180* (2001), 209–??

[68] Herman, T. Probabilistic self-stabilization. *Information Processing Letters 35*, 2 (1990), 63–67.
This paper proposes a probabilistic self-stabilizing algorithm for a unidirectional communication ring with identical processes. The number of processes is uneven. The algorithm circulates a single token in the ring. If the initial state of the ring is abnormal then the algorithm executes and the ring converges to a normal state with one token.

[69] Herman, T. Self-stabilization: Randomness to reduce space. *Distributed Computing 6*, 2 (1992), 95–98.

[70] Herman, T., and Pemmaraju, S. Error-detecting codes and fault-containing self-stabilization. *Information Processing Letters 73*, 1-2 (January 2000), 41–46.
This paper shows that fault containment, within a single step, is probabilistically achievable for many stabilizing programs without implying replication overhead. The model used in this paper for examining consists of a set of n processes that communicate using shared variables. Here a transformation procedure to convert a stabilizing program into a fault-containing stabilizing program is also introduced.

[71] Howell, R. R., Nesterenko, M., and Mizuno, M. Finite-state self-stabilizing protocols in message-passing systems. *Journal of Parallel and Distributed Computing 62*, 5 (May 2002), 792–817.

[72] Hsu, S.-C., and Huang, S.-T. Analyzing self-stabilization with finite-state machine model. In *Proceedings of the 12th IEEE International Conference on Distributed Computing Systems* (9-12 June 1992), IEEE, pp. 624–631.
This paper presents an approach to analyze the self-stabilizing algorithms with the finite state machine model. When a self-stabilizing algorithm is applied in a distributed system, a finite-state machine is used to model the behavior of each node. From the rules of the self-stabilizing algorithm, this approach defines some states and derives a state transition diagram. Then correctness of the algorithm can be proved and complexity can be analyzed.

[73] Hsu, S.-C., and Huang, S.-T. A self-stabilizing algorithm for maximal matching. *Information Processing Letters 43*, 2 (1992), 77–81.
Keywords: Distributed systems, fault-tolerance, maximal matching, self-stabilization, variant function.
This paper provides a self-stabilizing algorithm for finding a maximal matching in distributed networks. The distributed network is represented by an undirected graph $G = (V, E)$. Each node maintains a pointer and based on that pointer three possible states for each node are defined. A node having the privilege makes a move. After a finite number of steps the system finds itself in a legitimate state. The paper provides a variant function to prove the correctness of the algorithm. It also proves that the upperbound of the number of moves is $O(n^3)$.

[74] Huang, S.-T., and Chen, N.-S. A self-stabilizing algorithm for constructing breadth-first trees [4]. *Information Processing Letters 41*, 2 (February 1992), 109–117.

Keywords: Fault tolerance, self-stabilizing algorithms, breadth-first trees

This paper proposes a self-stabilizing algorithm for constructing breadth-first tree. The idea of self-stabilization given by Dijkstra [35] is slightly modified here. Once the system reaches the legimitate state, the algorithm is deadlocked. A connected graph $G = (V, E)$ is used to model the distributed system. From the graph a specific node r is selected as the root. Each node i other than r maintains two variables $L(i)$ and $P(i)$ representing its level and parent respectively. The algorithm has two rules and the processor(node) that satisfies any of the rules, when the system is in illegitimate state, enjoys the privilege and makes the move. There is a predefined predicate and the system reaches a legimitate state once this predicate is true. The algorithm, at this stage, constructs a breadth-first tree from G rooted at r and is deadlocked in the sense that no further computation step is carried out since no processor has the privilege.

[75] Huang, T. C., and Lin, J.-C. A self-stabilizing algorithm for the shortest path problem in a distributed system. *Computers and Mathematics with Applications 43*, 1-2 (January 2002), 103–109. Keyword: Central daemon, self-stabilizing algorithm, shortest paths, distance, turn nodes, bounded function, predecessors.

Huang and Lin proposes a self-stabilizing algorithm for finding the shortest paths from a node to each node in a distributed system. Two rules dominate the algorithm and make permissible changes in the value of a local variable of each node. The system eventually stabilizes in a legitimate state where each node's local variable contains the shortest path from the source.

[76] Huang, T. C., Lin, J.-C., and chen, H.-J. A self-stabilizing algorithm which finds a 2-center of a tree. *Computers and Mathematics with Applications 40*, 4-5 (August-September 2000), 607–624.

[77] Hutle, M., and Widder, J. Self-stabilizing failure detector algorithms. *IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN'05)* (February. 2005).

[4] Milestone paper

[78] Ishii, H., and Kakugawa, H. A self-stabilizing algorithm for finding cliques in distributed systems. In *Proceedings. 21st IEEE Symposium on Reliable Distributed Systems* (13-16 October 2002), IEEE, pp. 390–395.
A self-stabilizing algorithm for finding cliques in distributed systems is proposed in this paper. Each process p in the network G is given a set of its neighbor processes as input, and finds a set of its neighbors that are fully connected together with p. The algorithm consists of 4 guarded commands. The algorithm converges in $O(n^4)$ steps, where n is the number of processors.

[79] Kakugawa, H. Uniform and self-stabilizing fair mutual exclusion on unidirectional rings under unfair distributed daemon. *Journal of Parallel and Distributed Computing 62*, 5 (May 2002), 885–898.

[80] Karaata, M., and Chaudhuri, P. A dynamic self-stabilizing algorithm for constructing transport net. *Computing 68*, 2 (March 2002), 143–161.

[81] Karaata, M. H. A self-stabilizing algorithm for finding articulation points. *International Journal of Foundations of Computer Sciences 10*, 1 (1999), 33–46.
The self-stabilizing algorithm proposed in this paper finds the articulation points of a connected undirected graph. The algorithm uses the spanning tree constructed from breadth first search of the graph. The algorithm is based on the idea that a vertex v is an articulation point iff there exists two neighbors of node v in the spanning tree that are not transitively linked. The algorithm takes $O(n^2|E)$ moves to reach the legitimate state.

[82] karaata, M. H. A stabilizing algorithm for finding biconnected components. *Journal of Parallel and Distributed Computing 62*, 5 (May 2002), 982–999.
In order to find the biconnected components of a connected undirected graph, this paper proposes a self-stabilizing algorithm. The algorithm uses the spanning tree constructed by BFS in the graph. The algorithm bases on the idea that two fundamental cycles belong to the same biconnected component iff they are transitively connected. The proposed algorithm terminates after $O(d)$ time, where d is the diameter of the biconnected component with the largest diameter in the graph.

[83] Karaata, M. H., and Al-Anzi, F. S. A dynamic self-stabilizing algorithm for finding strongly connected components. Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing(PODC '99), pp. 465–470.

Keywords: Directed graphs, distributed systems, self-stabilizations, strongly connected components.
In this paper an optimal self-stabilizing algorithm is presented to find the strongly connected components of a directed graph. The algorithm is based on the idea that two nodes i, j of a graph belong to the same strongly connected component iff there exists a path from i to j and vice versa. The proposed algorithm takes $O(C)$ rounds to compute strongly connected components, where C is the length of the longest cycle in the graph.

[84] Karaata, M. H., and Chaudhuri, P. A self-stabilizing algorithm for bridge finding. *Distributed Computing 12*, 1 (March 1999), 47–53.
Keywords: Biconnected components, bridge, distributed systems, self-stabilization.
The algorithm proposed in this paper identifies the set of bridges in a connected undirected graph. A breadth first search is used to compute initially a BFS spanning tree of the graph. Then to detect the bridges in the graph a self-stabilizing algorithm is applied which is based on the idea that any edge (i, j) in the tree is not a bridge in the graph iff the graph has an edge (u, v) not in the tree where u is a descendent of an ancestor of i but not j and v is a descendent of j.

[85] Karaata, M. H., Pemmaraju, S. V., Bruell, S. C., and Ghosh, S. Self-stabilizing algorithms for finding centers and medians of trees. In *Symposium on Principles of Distributed Computing* (1994), p. 374.
Keywords: Center, distributed algorithm, median, self-stabilization, tree.
This paper presents a self-stabilizing algorithm to detect the centers and medians of trees. For each vertex two values - $h - value$ and $s - value$ are defined. to motivate the algorithm two conditions for these two values are given. A central scheduler arbitrarily selects an enabled guard and allows the execution of the corresponding atomic move to be completed, before any guard is re-evaluated. When all guards are false, the system reaches a state where the values satisfy their conditions. At this state, the vertex with maximum $h - value$ is the center and that with maximum $s - value$ is the median.

[86] Katz, S., and Perry, K. J. Self-stabilizing extensions for message-passing systems. In *Proceedings of the ninth annual ACM symposium on Principles of distributed computing* (1990), SIGOPS: ACM Special Interest Group on Operating Systems and SIGACT: ACM Special Interest Group on Algorithms and Computation Theory, ACM Press New York, NY, USA, pp. 91–101.

[87] Kruijer, H. S. M. Self-stabilization (in spite of distributed control) in tree-structured systems[5]. *Information Processing Letters 8* (January 1979), 91–95.
Keywords: Multiprocessing, networks, self-stabilization, distributed control, trees, equilibrium perturbation.
Kruijer presented a self-stabilizing algorithm for a distributed system to maintain the structure of a tree. A tree T with n nodes is considered. Each node represents a machine with even number(≥ 4) of states. The state of each node i representing a $2K$-state machine is defined by two variables $s[i](0, 1, \cdots K-1)$ and $eq[i]$ which is a boolean variable. Each machine can enjoy two privileges and the machine enjoying the privilege makes a corresponding move. This paper proves that regardless of the initial state and regardless of the privilege selected, the system reaches a legitimate state after a finite number of moves. The legitimate state is defined using the conditions of a perfect state. The states that arise from the permissible moves from a perfect state are also regarded as legitimate states.

[88] Kutten, S., and Patt-Shamir, B. Time-adaptive self stabilization. In *Symposium on Principles of Distributed Computing* (1997), pp. 149–158.
The basic problem of persistent bit, where the system is required to maintain a value in the face of transient failures by means of replication is considered in this paper. It proposes an algorithm to recover the value quickly. The algorithm can recover the value of the bit at all nodes in $O(f)$ time, where f is a transient fault hit.Moreover, complete state quiescence occurs in $O(d)$ time units, where d denotes the diameter of the network. The paper also gives a transformation procedure to convert a distributed non-reactive and non-stabilizing protocol into a self-stabilizing one.

[89] Lin, C., and Simon, J. Observing self-stabilization. In *Symposium on Principles of Distributed Computing* (1992), pp. 113–123.

[90] Lin, C., and Simon, J. Possibility and impossibility results for self-stabilizing phase clocks on synchronous rings. In *Proceedings of the Second Workshop on Self-Stabilizing Systems* (1995), pp. 10.1–10.15.

[91] Lin, J.-C., and Huang, T. C. An efficient fault-containing self-stabilizing algorithm for finding a maximal independent set. In *IEEE Transactions on Parallel and Distributed Systems* (August 2003), vol. 14, pp. 742–754.

[5]Milestone paper

[92] Mayer, Ostrovsky, and Yung. Self-stabilizing algorithms for synchronous unidirectional rings. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)* (1996).

[93] Mizuno, M., and Nesterenko, M. A transformation of self-stabilizing serial model programs for asynchronous parallel computing environments. *Information Processing Letters 66*, 6 (June 1998), 285–290.

[94] Nesterenko, M., and Mizuno, M. A quorum-based self-stabilizing distributed mutual exclusion algorithm. *Journal of Parallel and Distributed Computing 62*, 2 (February 2002), 284–305.

[95] Petit, F. Highly space-efficient self-stabilizing depth-first token circulation for trees. In *OPODIS'97, International Conference On Principles Of Distributed Systems Proceedings* (1997), pp. 221–235.

[96] Petit, F. *Efficiency and Simplicity in Self-Stabilizing Distributed Depth-First Token Circulation Algorithms*. PhD thesis, Universite de Picardie Jules Verne, TR98-05, LaRIA, Amiens France, 1998.

[97] Petit, F., and Villain, V. Color optimal self-stabilizing depth-first token circulation. In *PDCS-97 10th International Conference on Parallel and Distributed Computing Systems Proceedings* (1997), International Society for Computers and Their Applications, pp. 227–233.

In this paper a self-stabilizing depth-first token circulation protocol is proposed for uniform rooted networks. The contribution of the paper consists of explaining how the basic depth-first token circulation protocol is nearly self-stabilizing and how to obtain a self-stabilizing protocol by just adding what is necessary to destroy cycles. The proposed algorithm is very convenient to obtain the mutual exclusion or to construct a spanning tree.

[98] Prasetya, I., and Swierstra, S. Formal design of selfstabilizing programs. Tech. Rep. UU-CS-1995-07, Dept. of Computer Science, Utrecht University, 1995.

[99] Qadeer, S., and Shankar, N. Verifying a self-stabilizing mutual exclusion algorithm. In *IFIP International Conference on Programming Concepts and Methods (PROCOMET '98)* (Shelter Island, NY, 1998), D. Gries and W.-P. de Roever, Eds., Chapman & Hall, pp. 424–443.

[100] Schneider, M. Self-stabilization. *ACM Computing Surveys 25*, 1 (March 1993), 45–67.

This paper defines self-stabilization, examines its significance in the context of fault tolerance, defines the important research themes that have arisen from it, and discusses the relevant results. In addition to the issues arising from Dijkstras original presentation as well as several related issues, the paper discusses methodologies for designing self-stabilizing systems, the role of compilers with respect to self-stabilization, and some of the factors that prevent self-stabilization.

[101] Shi, Z., Goddard, W., and Hedetniemi, S. T. An anonymous self-stabilizing algorithm for 1-maximal independent set in trees. *Information Processing Letters 91*, 2 (July 2004), 77–83.

[102] Shukla, S., Rosenkrantz, D., and Ravi, S. Developing self-stabilizing coloring algorithms via systematic randomization. In *Proceedings of the International Workshop on Parallel Processing* (Bangalore, India, 1994), Tata-McGrawhill, New Delhi, pp. 668–673.

[103] Sur, S., and Srimani, P. K. A self-stabilizing distributed algorithm to construct BFS spanning trees of a symmetric graph. *Parallel Processing Letters 2*, 2-3 (1992), 171–179.

The self-stabilizing algorithm proposed in this paper constructs a BFS spanning tree of an arbitrary connected symmetric graph. The algorithm has a single uniform rule to assign privilege to a node and make it move. Whenever the system is in an illegitimate state at least one of the nodes should be able to recognize it and should take some action. The paper provides a correctness proof based on graph theoretical reasoning that can also be used for proving the correctness of other self-stabilizing algorithms.

[104] Theel, O., and Gärtner, F. C. On proving the stability of distributed algorithms: self-stabilization vs. control theory. In *Proceedings of the International Systems, Signals, Control, Computers Conference (SSCC'98), Durban, South Africa* (1998), V. B. Bajic, Ed., vol. III, pp. 58–66.

[105] Varghese, G. Self-stabilization by counter flushing. In *Symposium on Principles of Distributed Computing* (1994), pp. 244–253.

[106] Xu, Z., Hedetniemi, S. T., Goddard, W., and Srimani, P. K. A synchronous self-stabilizing minimal domination protocol in an arbitrary network graph. In

Proceedings of the 5th International Workshop on distributed computing(IWDC) (27-30 December 2003), LNCS 2918, pp. 26–32.

[107] YAHFOUFI, N., AND DOWAJI, S. Self-stabilizing distributed branch-and-bound algorithm. In *Proceedings of the 1996 IEEE 15th Annual International Phoenix Conference on Computers and Communications* (1996), pp. 246–252.

[108] YEN, H.-C. Analysis of self-stabilization for infinite-state systems. In *Proceedings. Seventh IEEE International Conference on Engineering of Complex Computer Systems* (11-13 June 2001), IEEE, pp. 240–248.

[109] YEN, I.-L. A highly safe mutual exclusion self-stabilizing algorithm. *Information Processing Letters 57*, 6 (March 1996), 301–305.

[110] Saifullah, A. and Tsin, Y.-H.; Self-stabilizing Computation of 3-edge-connected Components; International Journal of Foundation of Computer Science, vol. 22, No. 5, pp. 1161--1185; 201

[111] Saifullah, A. and Tsin, Y.-H.; A Self-stabilizing Algorithm for 3-edge-connectivity; International Journal of High Performance Computing and Networking, Vol. 7. No. 1, pp. 40--52; 2011

[112] Saifullah, A. and Tsin, Y.-H.; A Self-stabilizing Algorithm for 3-edge-connectivity; The 5th International Symposium on Parallel and Distributed Processing and Applications (ISPA 2007)