Chapter 5

Self–Stabilizing Graph Coloring Algorithms

Shing-Tsaan Huang
National Central University, Taiwan

Chi-Hung Tzeng
National Tsing Hua University, Taiwan

Jehn-Ruey Jiang
National Central University, Taiwan

ABSTRACT

The concept of self-stabilization in distributed systems was introduced by Dijkstra in 1974. A system is said to be self-stabilizing if (1) it can converge in finite time to a legitimate state from any initial state, and (2) when it is in a legitimate state, it remains so henceforth. That is, a self-stabilizing system guarantees to converge to a legitimate state in finite time no matter what initial state it may start with; or, it can recover from transient faults automatically without any outside intervention. This chapter first introduces the self-stabilization concept in distributed computing. Next, it discusses the coloring problem on graphs and its applications in distributed computing. Then, it introduces three self-stabilizing algorithms. The first two are for vertex coloring and edge coloring on planar graphs, respectively. The last one is for edge coloring on bipartite graphs.

INTRODUCTION

A distributed system can be considered as a network of nodes. Each node has program code, constants and variables. Over-heating, electromagnetic radiation or others may cause faults on a node. A node can easily restore its program code and constants by fetching them from a secondary storage. Or, the program code and constants can be burned in Read-Only-Memory. Therefore, program code and constants can be easily recovered from faults. However, variables which reflect the state of a node must be kept in Random-Access-Memory, and hence are vulnerable to faults. Here in this chapter, we refer faults that only affect the values of variables as transient faults. A transient fault on a node may make a variable perturbed to be any possible value. For example, a variable of
3 bits may be any value between 0 and 7 after a transient fault.

The system configuration of a distributed system consists of states of all the nodes and hence is reflected by the variables maintained by the nodes. Joining or leaving of nodes from the system changes the system configuration. Therefore, we also model transient faults to include the changes of system configurations, such as joining or leaving of nodes from the system.

The term self-stabilization in distributed computing was introduced by Dijkstra (Dijkstra, 1974). The states of a distributed system can be divided into two categories: legitimate states and illegitimate states. Ideally, the system should remain in legitimate states to work properly, but unexpected transient faults may bring it into an illegitimate state. A system is said to be self-stabilizing if (1) it can converge to a legitimate state regardless of any initial (possibly illegitimate) state in finite time, and (2) when it is in a legitimate state, it remains so henceforth. That is, a self-stabilizing system guarantees to converge to a legitimate state in finite time no matter what initial state it may start with; or, it can recover from transient faults automatically without any outside intervention.

As mentioned above, a distributed system can be considered as a network of nodes which can be represented by a graph \((V, E)\), where \(V\) is the set of nodes and \(E\) is the set of links. Two nodes are said to be each other’s neighbor if there is a link between them.

A self-stabilizing algorithm usually consists of several rules for each node. Each rule has the form: guard \(\rightarrow\) move. The guard is a boolean function depending on the state of the node and the states of its neighbors. And, the move part only changes the state of the node by referencing the state of itself and the states of its neighbors. We say a node has a privilege to execute the move part of a rule when the guard of the rule is evaluated to be true. However, to be able to really make a move, the node must be scheduled to execute the move.

There are several execution models (Wuu and Huang, 1995) discussed in the literature for self-stabilization. In this chapter, we only introduce the simplest one, i.e., the central daemon model. This model assumes a centralized scheduler called the central daemon. The central daemon arbitrarily selects one at a time of the privileged nodes to make a move. After the selected node finishes the move, the daemon again chooses a privileged node, if any, to take another move, and so on. The moves of the nodes are hence serialized. Therefore, it is also called the serial execution model. The selection by the central daemon is unpredictable, so the execution sequence can be in any order.

A computation of the system is then a sequence of moves of the nodes. It is usually not an easy task to design a self-stabilizing system because starting from any arbitrary initial state or any possible state after a transient fault, the system must have the ability to bring itself into a legitimate state in any possible execution sequence of finite moves and remains hereafter in legitimate states if no transient fault occurs.

Now, we show a trivial example to see how a self-stabilizing system works. Consider a system with five nodes connected as a ring. Each node \(x\) (\(0 \leq x \leq 4\)) maintains a variable \(S.x\) with the range \([0..3]\). The system is said to be in a legitimate state when all \(S.x\) have the same value, and in an illegitimate state otherwise. With an arbitrary initialization or after a transient fault, each \(S.x\) may have a value of 0, 1, 2, or 3.

There is only one rule for each node.

**Rule:** \(S.x \neq \max(S.x \mid x \in N(x)) \rightarrow S.x = \max(S.x \mid x \in N(x))\),

where \(N(x)\) is the set consisting of neighbors of node \(x\) and node \(x\) itself.

Intuitively, the rule makes each node set the value of its variable \(S\) to be equal to the largest one in its neighborhood. Therefore, all nodes will eventually have the same variable \(S\) value to reach