Memory efficient, self-stabilizing algorithm to construct BFS spanning trees *

Colette Johnen

L.R.I./C.N.R.S. URA 410, Université Paris XI, Bat. 490, F-91405 Orsay Cedex, France.
colette@lri.fr, http://www.lri.fr/~colette/

The spanning tree construction is a fundamental task in communication networks. Improving the efficiency of the underlying spanning tree algorithm usually also corresponds to the improvement of the efficiency of the entire system. One of the important performance issues of self-stabilizing algorithms is the memory requirement per processor. The self-stabilizing spanning-tree algorithms to-date need a *distance* variable, which keeps track of the current level of the processor in the BFS tree. Thus, these BFS spanning tree construction algorithms have the space complexity of at least $O(\log N)$ bits per processor, where N is the number of processors. Some authors have proposed specific data structures to store the *distance* variable in a distributed manner, thus reducing the memory requirement.

We present a self-stabilizing BFS spanning tree construction algorithm which requires only $O(1)$ bits of memory per link. The algorithm uses neither the *distance* variable nor any special data structure to achieve the memory requirement. One of the desirable features of the protocols written in large distributed systems is that the cost does not depend on the global properties, such as network size, which can change over time. Our algorithm has this feature: when the network size changes, the algorithm does not need to be modified. The code at a processor needs to be modified only when the degree of the processor changes (a locally checkable property).
It is known that no deterministic algorithm can construct a spanning tree in an anonymous (uniform) network. The best that can be proposed is a semi-uniform deterministic algorithm as ours, in which all processors except one execute the same code. We call the distinguished processor the *legal root* (also denoted r) which eventually becomes the root of the BFS trees.

Each processor i maintains the following variables: (i) TS and P: Pointers to one of its neighbors (called i's parent) or to $NULL$; (ii) C: Color of $i \in \{0, 1\}$; (iii) S: Status of $i \in \{Idle, Working, Power, Erroneous\}$; (iv) ph: Phase of $i \in \{a, b\}$.

Our algorithm is a non-terminating algorithm. The legal root alternately builds 0-colored and 1-colored BFS spanning trees (the obtained trees may differ from a construction to another one). We use r_color to denote the color of the current tree. The color is used to distinguish the processors that are from those that are not part of the current tree: only the processors in the tree have r_color.

A difficulty is to build BFS trees without using the *distance* variable: to ensure that the path of each processor to r in the obtained tree is minimal. Our solution is to build the trees in phases: during a kth phase, all processors at a distance of k from r join the tree by choosing a neighbor having the $Power$ status as a parent. (they update their P and TS variables). Only the leaves (processors at a distance of k-1 from r) take the $Power$ status, other processors in the tree take the $Working$ status. The processors change to $Idle$ status when the current phase is over in their neighborhood: their neighbors have r_color. The legal root detects the end of the current phase and initializes a new one by changing its phase value. When the construction of the sub-tree rooted to a processor is over the processor sets its P variable to $NULL$. At the end of the tree construction, the tree structure is stored in the TS variables. Our algorithm implements a centralized algorithm (requiring the knowledge of all processor states by r to decide the beginning of a new tree construction or of a phase) in a distributed fashion where processors have only a partial view of the system state.

We have designed two error-handling strategies: one for destroying illegal trees and the other for breaking the cycles. Illegal roots detect their abnormal situation and take the $Erroneous$ status. The $Erroneous$ status is propagated to their leaves. The $Erroneous$ leaves are detached from their branch. The repetition of detaching and recovering processes will correct all processors inside the illegal trees.

Our strategy for breaking cycles is different from the previous solutions. algorithm will create an abnormal situation in the neighborhood of cycles. The abnormal situation is detected by a processor inside the cycle and this processor initiates the cycle destruction process. A processor having a parent assumes that it is in the legal tree and has r_color. Based on this assumption, it detects a conflict when a neighbor with $Power$ status does not have its color (both cannot be inside the legal tree). Once a processor detects a conflict, it eventually chooses the processor with $Power$ status as the parent and the cycle is broken. During a 0-colored (1-colored) tree construction, the tree expands until it reaches the first 1-colored (0-colored) cycle, if it exists. This cycle will transform itself into a branch of the legal tree. As the processors inside a cycle cannot change their color, the cycles are eventually destroyed.

*A large version is in the Proc. of the Third Workshop on Self-Stabilizing Systems, Santa-Barbara, CA, Aug. 1997.