Memory-Efficient Self-Stabilizing Algorithm to Construct BFS Spanning Trees

Colette Johnen

L.R.I./C.N.R.S., Université de Paris-Sud
Bat. 490, Campus d'Orsay
F-91405 Orsay Cedex, France.
colette@lri.fr
http ://www.lri.fr/~colette/

Abstract. In this paper, we consider the problem of fault-tolerant distributed BFS spanning tree construction. We present an algorithm which requires only $O(1)$ bits of memory per incident network edge on an uniform rooted network (no identifier, but a distinguished root). The algorithm works even in the case of worst transient faults (i.e., it is self-stabilizing).

Keywords: Mutual exclusion, self-stabilization, BFS spanning tree.

1 Introduction

In this paper, we are interested in the self-stabilizing construction of a BFS (Breadth-First Search) spanning tree on an uniform rooted network (no identifier, but a distinguished root). A spanning tree is breadth-first provided that each processor at (shortest) distance d from the root in the original graph appears at depth d in the tree (i.e., at a distance d from the root in the tree). The spanning tree construction is a fundamental task in communication networks. Many crucial network tasks, such as network reset (and thus any input/output task), leader election, broadcast, topology update, and distributed database maintenance, can be efficiently carried out in the presence of a spanning tree. Thus, improving the efficiency of the underlying spanning tree algorithm usually also means the improvement of the efficiency of the particular task.

Self-stabilizing spanning trees have been presented in [AKY90], [CYH91], [HC92], [SS92], [DIM93], and [TH94]. Dolev presented a time optimal BFS tree protocol in [Dol93]. In [BLB95], Butelle, Lavault, and Bui reported a minimum diameter spanning tree algorithm. A BFS spanning tree algorithm on unidirectional network is presented by Afek and Bremler in [AB97]. In all these algorithms, each processor has 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, N being the number of processors.

There are two main parameters to measure the efficiency of self-stabilizing algorithms: stabilization time, and memory requirement per processor. In large

distributed networks (containing several millions of processors) managed by several organizations, the properly functioning of network management protocols should not depend on global properties (such as network size) which can be modified at any time. Therefore, we propose an algorithm whose space complexity is independent of the network size ($O(1)$ bits per edge). When, the network grows, the code does not need to be changed. The code at a processor needs be modified only when the degree increases (a locally checkable property). Awerbuch and Ostrovsky [AO94] proposed a $log^*(N)$ data structures in a distributed manner. Itkis and Levin presented, in the appendix of [IL94], another data-structure which is based on Thue-Morse sequence requiring $O(1)$ bits per edge.

We develop a completely new approach to design our algorithm where the *distance* variable is not required (BFS trees are built in phases). Also, the space complexity of our algorithm is intrinsically $O(1)$ bits per edge without using any special data structures.

The paper is organized as follows: The formal model is described in Section 2. A detailed description of the algorithm is given in Section 3, its correctness is discussed in Section 4.

2 Model

The chosen computation model is an extension of Dijkstra's original model for rings to arbitrary graphs [Dij74]. Consider a symmetric connected graph G(V, E), in which V is a set of processors and E is a set of symmetric edges. We use this graph to model a distributed system with N processors, $N = |V|$. In the graph, the directly connected processors are called neighbors. Each processor i maintains a set of neighbors, denoted as NB_i. A processor state is defined by its variable values. Each processor has a single-writer, multi-reader *register*. A processor only communicates with its neighbors by using its *register* which it writes its state into and all its neighbors read from. The system state is defined by the set of processor states.

The proposed self-stabilizing algorithm consists of a set of rules. Each rule has two parts: the privilege (condition) and the move. The privilege is defined as a boolean function of the processor's own state and the state of its neighbors. When the privilege of a rule on a processor is true, we say that the processor has the privilege. A processor having a privilege may then, but it *does not need*, make the corresponding move which changes the processor's state into a new one (and updates its register). The rules are assumed to be executed atomically: processors cannot evaluate their privilege and then make the corresponding move later in another atomic step.

A *computation* is defined to be a sequence of system states (s_1, s_2, ... , s_n, ...) where every pair of states (s_i, s_{i+1}) is a computation step. In a computation step, several processors (at least one) execute a move (and update their registers). During a step, a processor may execute at most one move (non-deterministically chosen) even if it satisfies several privileges. When a processor

holds a privilege without executing its rule forever, the computation is said to be *unfair*. An unfair computation may lead to a situation where a section of the code of a processor is no more executable. As self-stabilizing systems do not cope with program corruptions, we exclude unfair computations in this work.

A *region* of a system is a subset of system states. A region A is *closed* if no move allows the system to quit the region. A computation $\mathcal{C}$ *reaches* a region A, if $\mathcal{C}$ has a state in A. A region $A1$ is an *A0-attractor*, if $A1$ is closed and all computations whose initial state is in $A0$, reach $A1$. A system *self-stabilizes* to A if and only if regardless of the initial state and regardless of the computation, the system is guaranteed to reach A, after a finite number of moves and A is closed. In fact, A is an $A0$-attractor of the system ($A0$ being the set of system states).

3 Algorithm Specification

We present an anonymous algorithm that builds a BFS spanning tree. Angluin [Ang80] has shown 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 r, the distinguished processor, the *legal root*, which will eventually be the root of the BFS tree.

During the stabilization period, illegal branches or cycles may exist, and a tree construction may only be partial. Therefore, the algorithm is non-terminating— at the end of a tree construction, the legal root initiates the same process. It builds 0-colored and 1-colored BFS spanning tree alternately The color is used to distinguish the processors of the tree from those that are not part of the tree: the processors in the tree only have the tree color, named : r_color.

The main difficulty to build a BFS tree without using the *distance* variable is to ensure that the path of each processor to r in the tree is minimal. Our approach is to build the tree in phases: during the kth phase, all processors at a distance of k from r join the tree (by choosing a processor at a distance $k-1$ from r as parent) r detects the end of a phase and initiates new phases. Our algorithm actually implements a centralized algorithm, requiring the knowledge of all processor states by the legal root to decide the initiation of a new tree construction or of a phase, in a distributed fashion where processors have only a partial view of the system state.

There are two major error handling tasks: one to remove the illegal branches and the other to break the cycles. Our approach to handling the illegal branches (which are not cycles and are not rooted at the legal root) is similar to the ones in [HC93] and [JB95]. The illegal roots detect their abnormal situation and take *Erroneous* status. *Erroneous* status is propagated to their leaves. *Erroneous* leaves are detached from their branch. Finally, these detached processors are recovered.

The cycle elimination strategy is similar to the one proposed in [JB95]. Typically, a *distance* variable is used for this purpose. But, we do not use such a variable. The basic idea is that the process of tree construction will create an abnormal situation in the neighborhood of cycles, and the abnormal situation will be detected by a processor inside the cycle which then initiates the cycle destruction process.

We have four sets of rules : the rules R0-R6 designed to ensure the tree constructions, presented in section 3.2; the rules R7-R11 designed to ensure the illegal trees destruction, explained in section 3.3; the rules R12 and R13 designed to break cycles are detailed in section 3.3; and finally, the rule R14 designed to handle faulty processors, in section 3.3.

3.1 Variables

Each processor i maintains the following variables ($X.i$ denotes X of i and $X.Y.i$ denotes X of Y of i):

$TS.i$	The parent pointer pointing to one of its neighbors or containing $NULL$. TS variables maintain the tree structure in a distributed manner.
$P.i$	The parent pointer pointing to one of its neighbors or containing $NULL$. When $P.i \neq NULL$, $P.i = TS.i$. When a correct tree is created, $P.i = NULL$. The variable P is used to decide whether the tree construction is complete or not—if at end of a phase, i has no child, (i.e., a neighbor whose P variable points to i), then the tree construction is done in i's subtree.
$C.i$	The color which takes value from the set $\{0, 1\}$. Once the system stabilizes, the processors in the current tree have r_color while other processors have the complement of r_color.
$S.i$	The status which takes value from the set $\{Idle, Working, Power, Erroneous\}$. The processors having $Power$ status only can create new children. The processors at a distance of $k-1$ from r only will acquire this status during the kth phase. Once the system stabilizes, if the current phase is begun in its subtree and not yet terminated, then a processor has $Working$ status. A processor is $Idle$, if it is not inside the tree, or if its subtree finished a phase, but did not start the next phase. $Erroneous$ status is used during the error recovering process.
$ph.i$	The phase which takes value from the set $\{a, b\}$. The value of S does not indicate if the current phase is done or not. A processor in the tree is $Idle$ when it has completed or has not started the current phase. In order to distinguish between these two cases, we use the phase variable. If the phase value of an $Idle$ processor is the same as that of its parent, then the $Idle$ processor has finished the current phase. Otherwise, it has not initiated the current phase.

The root r maintains the same variables, except P and TS—r does not have a parent. And $S.r$ cannot have the value $Idle$.

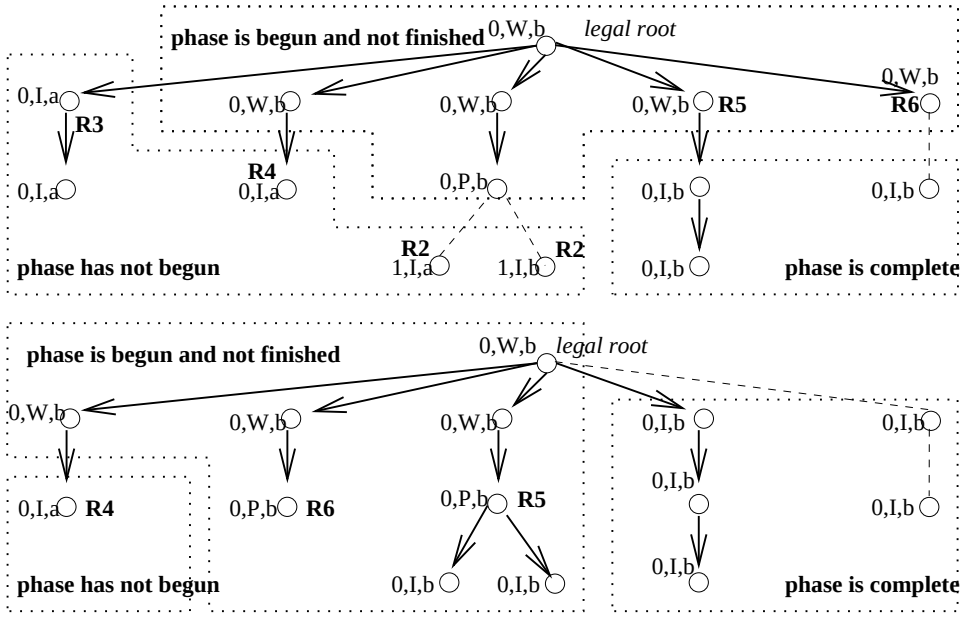
The size of P and TS of a processor i is $log(\triangle_i)$ where $\triangle_i$ is the degree of i. The color, status, and phase variables have a constant size (total 4 bits). Thus, the space complexity of the algorithm is $O(\log(\triangle))$ (i.e., $O(1)$ bits per edge).

During each computation step, a processor computes its children set ($i \in Child.i \Leftrightarrow P.i = j$) according to the value of P of its neighbors.

3.2 Tree construction rules

The rules R0 to R6 have been designed to ensure the tree constructions. A R0 move initiates the tree constructions; a R1 move initiates the phases (R0 and R1 may only be performed by r). R3 and R4 moves propagate the phase wave from r to the processors in tree. Processors joint the legal tree by a R2 move. R5 and R6 move propagates the termination signal of the current phase to r.

Fig. 1. A computation step during the 3rd phase of a 0-colored tree construction (after stabilization).

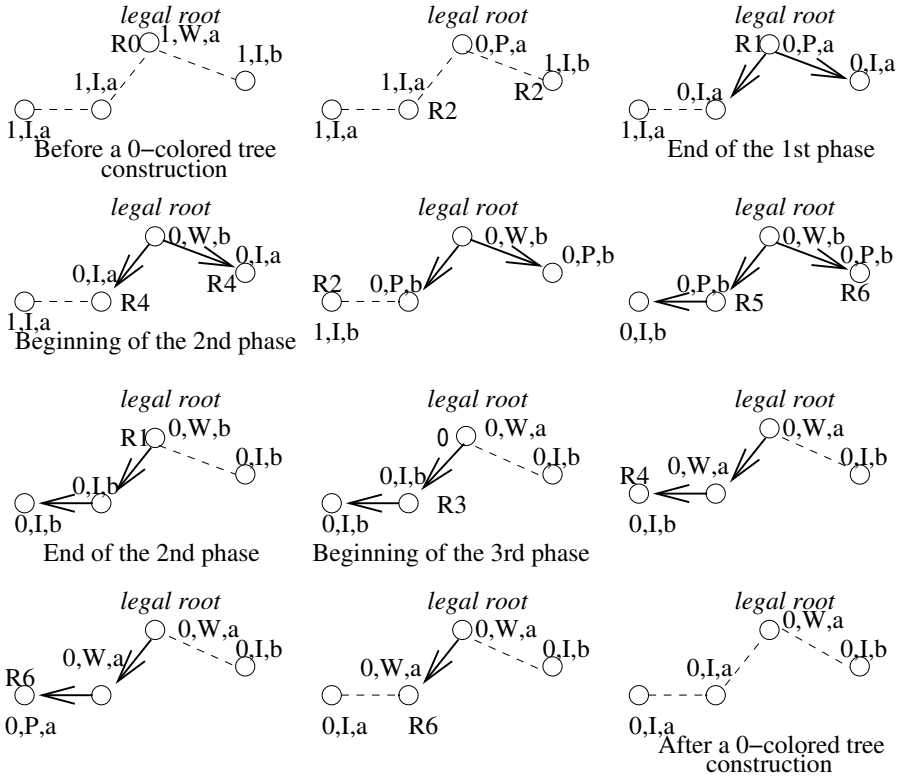


After the execution of R2, R3, R4, R5, and R6 moves.

In the beginning of the kth phase (see Figure 1), all processors in the tree take $Working$ status and r's phase value (by a R3 move), except the leaves

(processors at a distance of $k-1$ from r) which take *Power* status (by a R4 move). All processors at a distance of k from r join the tree by choosing a *Power* status neighbor as a parent (update their P and TS variables, but also take the phase value and color of the new parent) by a R2 move. The processors with *Power* status will finish the kth phase (change their status to *Idle*) when the current phase is over in their neighborhood: all their neighbors are in the tree (they have *r_color*) by a R5 or R6 move. The *Working* processors will finish the phase when their children have finished the current phase (they are *Idle* and have the same phase value as theirs) by a R5 or R6 move.

Fig. 2. An 0-colored tree construction



The rule R0 initiates a tree construction (see Figure 2). r changes its color and initiates the first phase (by taking the *Power* status). All r's neighbors join the tree by executing R2. When the current phase (say $k-1$) is over, r initiates the next one by changing its phase value by executing R1. R3 propagates the phase value in the tree. R4 assigns *Power* status to the processors at a distance of k-1 from r. By R2 moves, all the neighbors of the current leaves join the tree (which are at a distance of k from r). R5 propagates the termination signal of

the current phase to r. When the kth phase is over (i.e., the processors at a distance of k from r are in the tree, and the processors inside the tree are $Idle$ and have the same phase as that of r's), r initiates the $k+1$ phase by executing R1. When the subtree rooted at a processor is complete (i.e., its Child set is empty), the processor sets its P variable to $NULL$ by executing R6. Thus, when r becomes childless (child.$r = \oslash$), the tree construction is complete. The tree is stored locally in the TS variables. r initiates a new tree construction, by a R0 move.

We define some predicates which are used in the algorithm.

- Conflict$(i, k) \equiv [\ (P.i \neq NULL) \wedge (k \in NB_i) \wedge (S.k = Power) \wedge (C.k \neq C.i) \wedge$
  $(\ (S.i \neq Erroneous) \vee (P.i \neq k)\ )\ ]$
  The processor i has a neighbor k which has an "unexpected" color with respect to i. Since i has a parent and k has the $Power$ status, both should be in the legal tree and should have the r_color. But, in this case, i has a neighbor k with $Power$ status and with a different color than of i. Moreover, i does not have $Erroneous$ status or is not a child of k. Thus, i may be inside a cycle.
- Connection$(i, k) \equiv [\ (S.i = Idle) \wedge (P.i = NULL) \wedge (Child.i = \oslash) \wedge$
  $(k \in NB_i) \wedge (C.k \neq C.i) \wedge (S.k = Power)\ ]$
  k has $Power$ status and does not have the color of i. i is an $Idle$ processor with no parent and no child. Therefore, i assumes that k is a leaf of the current legal tree with $Power$ status. Thus, i will eventually join the tree during the current phase by choosing k as the parent and taking r_color as its color.
- NewPhase$(i) \equiv [\ (i \neq r) \wedge (P.i \neq NULL) \wedge (S.P.i = Working) \wedge$
  $(C.P.i = C.i) \wedge (ph.i \neq ph.P.i) \wedge (S.i = Idle)\ ]$
  i's parent has begun a phase, but i has not. i is an $Idle$ processor, it has a parent with $Working$ status, and i's phase differs from its parent phase, but they have the same color.
- EndFirstPhase$(i) \equiv [\ (S.i = Power) \wedge\ (\ \forall j \in NB_i, C.j = C.i\ ) \wedge$
  $(\ \forall j \in Child.i, (S.j = Idle) \wedge (ph.j = ph.i)\ )\ ]$
  When i has the $Power$ status, all its neighbors must join the tree, and thus, take r_color (which is also the color of i). Therefore, i terminates this phase when its neighbors have its color. For error-recovering purpose, i will only finish this phase when its children are $Idle$ and they have the same phase value.
- EndPhase$(i) \equiv (S.i = Working) \wedge$
  $[\ (\ \forall j \in Child.i, (S.j = Idle) \wedge (ph.j = ph.i)\ )\ ]$
  i has finished a phase where it did not have $Power$ status. The children of i have finished the current phase— they are $Idle$ and have the same phase value as i.
- EndLastPhase$(i) \equiv [\ (Child.i = \oslash) \wedge (\ \text{EndFirstPhase}(i) \vee \text{EndPhase}(i)\ ) \wedge$
  $(\ \forall j \in NB_i, \neg\text{Conflict}(i, j)\ )\ ]$
  i has finished a phase and i has no more child. The current tree construction terminates in the subtree of i.

- EndIntermediatePhase(i) $\equiv$ [$(Child.i \neq \oslash) \wedge$ (EndFirstPhase(i) $\vee$ EndPhase(i))]
  i has finished a phase and it still has some children. The tree construction is not over in its subtree.

Fig. 3. Rules for the tree construction.

R0 : EndLastPhase(i) $\wedge$ $i = r$ $\rightarrow$ $C.r = (C.r + 1) mod 2$; $S.r = Power$.

R1 : EndIntermediatePhase(i) $\wedge$ $i = r$ $\rightarrow$ r changes its phase value ; $S.r = Working$.

R2 : Connection(i, k) $\wedge$ $i \neq r$ $\rightarrow$ $C.i = C.k$; $ph.i = ph.k$; $S.i = Idle$; $P.i = k$; $TS.i = k$.

R3 : NewPhase(i) $\wedge$ $Child.i \neq \oslash$ $\rightarrow$ $ph.i = ph.P.i$; $S.i = Working$.

R4 : NewPhase(i) $\wedge$ $Child.i = \oslash$ $\rightarrow$ $ph.i = ph.P.i$; $S.i = Power$.

R5 : EndIntermediatePhase(i) $\wedge$ $i \neq r$ $\rightarrow$ $S.i = Idle$.

R6 : EndLastPhase(i) $\wedge$ $i \neq r$ $\rightarrow$ $S.i = Idle$; $P.i = NULL$.

3.3 Error Handling Rules

A distributed system has an unpredictable initial state. Initially, the parent pointers may point to any neighbor or $NULL$. Thus, illegal trees (trees whose roots are not r) and cycles (paths without a root) may exist in the initial state. We propose two error-handling strategies: one for eliminating the illegal trees and the other for breaking the cycles.

Illegal Tree Elimination Rules The rules R7 to R11 have been designed to ensure the illegal trees destruction. R7 detects the illegal roots (processors that are root of an illegal tree). R8 (R11) propagates the *Erroneous* status forward in the trees (backward). R9 detaches *Erroneous* leaves from their branch. R10 recovers processors having the *Erroneous* status.

The elimination of illegal trees has been reported in [HC93] and [JB95]). The illegal roots detect the abnormal situation and take *Erroneous* status by a R7 move. *Erroneous* status is propagated to their leaves by a series of R8 moves. Then, *Erroneous* leaves quit their branch by a R9 move and become detached (the illegal trees still have *Erroneous* leaves though). Finally, the detached (i.e., without a parent and child) *Erroneous* processors are recovered: the change their status by executing R10. The repetition of detaching and recovering process will correct all processors inside the illegal trees (see an example in Figure 5).

The current phase never ends on the branches having an *Erroneous* processor. Therefore, when the legal tree has an *Erroneous* processor, its construction is locked: the legal tree has to be destroyed to avoid a deadlock. In this case, by

R11 moves the legal root will get the *Erroneous* status. Then by R8, R9, and R10 moves, the tree will delete itself.

We will define some predicates which will be used in the of illegal tree elimination rules.

- Detached(i) $\equiv \big[\ (Child.i = \oslash) \wedge (P.i = NULL)\ \big]$ i has no child and no parent.
- IllegalRoot(i) $\equiv \big[\ (i \neq r) \wedge (P.i = NULL) \wedge$
  $(\ (Child.i \neq \oslash) \vee (S.i = Working) \vee (S.i = Power)\ )\ \big]$
- ErroneousLeaf(i) $\equiv \big[\ (Child.i = \oslash) \wedge (P.i \neq NULL) \wedge (S.i = Erroneous) \wedge$
  $(S.P.i = Erroneous) \wedge\ (\ \forall j \in NB_i, \neg\text{Conflict}(i,\ j)\ )\ \big]$

Fig. 4. Rules for Illegal Tree Elimination.

R7 : IllegalRoot(i) $\wedge\ S.i \neq Erroneous\ \rightarrow\ S.i = Erroneous.$

R8 : $S.P.i = Erroneous \wedge S.i \neq Erroneous\ \rightarrow\ S.i = Erroneous.$

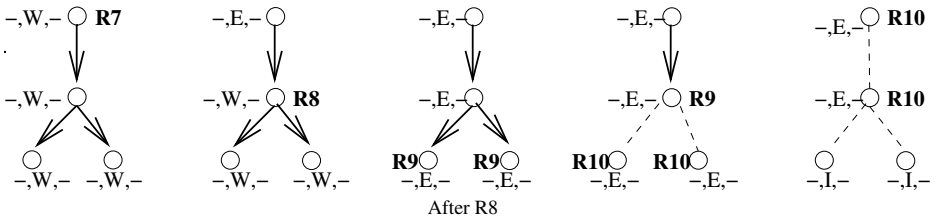
R9 : ErroneousLeaf(i) $\rightarrow\ P.i = NULL.$

R10a : Detached(i) $\wedge\ i \neq r \wedge S.i = Erroneous\ \rightarrow\ S.i = Idle.$

R10b : Detached(i) $\wedge\ i = r \wedge S.i = Erroneous\ \rightarrow\ S.i = Working.$

R11 : $S.i \neq Erroneous \wedge (\exists j \in Child.i,\ S.j = Erroneous)\ \rightarrow$
$S.i = Erroneous.$

Fig. 5. Illegal Tree Elimination.



Cycle destruction rules The rules R12 and R13 have been designed to ensure the cycles destruction. The rule R12 detects the conflicts (and thus cycles); the rule R13 breaks the cycles.

A processor having a parent assumes that it is in the legal tree and its color is equal to r_color (even if it is inside a cycle). Based on this assumption, it detects a conflict when a *Power* neighbor does not have its color (both cannot be inside the legal tree). When a processor detects a conflict, it will eventually choose the *Power* processor as the parent, and the cycle will be broken (transformed into a branch of the legal tree). In terms of the rules, when a processor detects a

conflict, it takes the *Erroneous* status by executing R12 (Figure 7). *Erroneous* status is propagated to the descendants/ancestors of the processor by a series of R8 moves. When it has an *Erroneous* parent and only *Erroneous* children, it chooses the *Power* processor as the parent by a R13 move, and the cycle is transformed into a branch of the legal tree. A series of R11 moves propagates the *Erroneous* status toward the legal root. Once the legal root has the *Erroneous* status, the legal tree will delete itself (just like the illegal trees).

During a 0-colored (1-colored) tree construction, the tree grows 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. Since the processors inside a cycle cannot change their color, the cycles are eventually destroyed.

The following predicate is used in the cycle destruction rules:

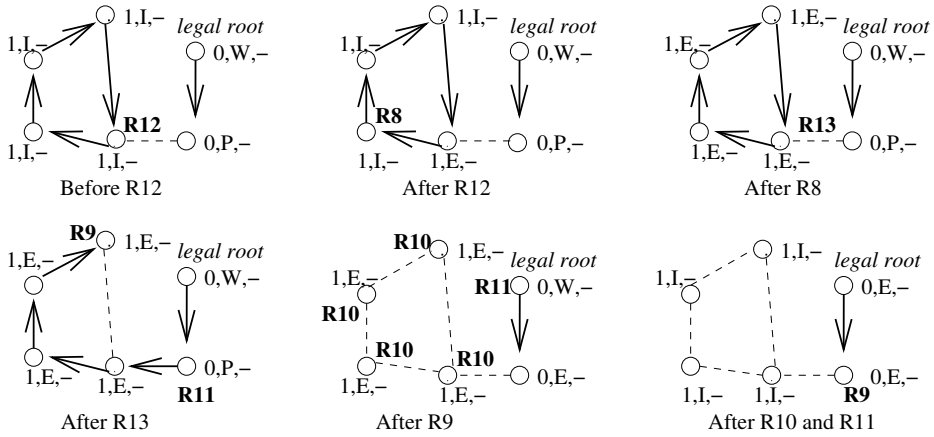
- ErroneousConflict$(i,\ k) \equiv [\ \text{Conflict}(i,\ k) \wedge S.P.i = Erroneous \wedge S.i = Erroneous \wedge (\ \forall j \in Child.i, S.j = Erroneous\ )\ ]$
  i is in conflict with k. i, i's parent, and i's children have the *Erroneous* status.

Fig. 6. Rules for Cycle Elimination.

R12 : Conflict$(i,\ k) \wedge S.i \neq Erroneous\ \rightarrow S.i = Erroneous.$

R13 : ErroneousConflict$(i,\ k) \rightarrow\ P.i = k.$

Fig. 7. Cycle Elimination.



Miscellaneous error recovering rule A processor is *Faulty* if it does not have the right color, the right status, or the right phase according to its parent's

state. For instance, a processor verifies the Faulty predicate if its color differs from its parent's (except if the processor or its parent has *Erroneous* status). In fact, all processors inside the legal tree should have color equal to r_color. Other rules are as follows: a $Power$ or a $Working$ processor should have a parent having *Erroneous* or $Working$ status. If a processor and its parent are *Idle* or $Working$, they should have the same phase value. A *Power* processor should have the same phase value as its parent has. The *Idle* children of a *Power* processor should have the same phase value as their parent has and should have no children.

- $\text{Faulty}(i) \equiv [\ (P.i \neq NULL) \wedge$
  $(\ [\ (C.i \neq C.P.i) \wedge (S.i \neq Erroneous) \wedge (S.P.i \neq Erroneous)\ ] \vee$
  $[\ (S.P.i = Idle) \wedge (\ (S.i = Working) \vee (S.i = Power)\ )\ ] \vee$
  $[\ (S.P.i = Power) \wedge (\ (S.i = Working) \vee (S.i = Power)\ )\ ] \vee$
  $[\ (S.P.i = S.i) \wedge (\ (S.i = Idle) \vee (S.i = Working)\ ) \wedge$
  $(ph.i \neq ph.P.i)\ ] \vee$
  $[\ (S.P.i = Working) \wedge (S.i = Power) \wedge (ph.i \neq ph.P.i)\ ] \vee$
  $[\ (S.P.i = Power) \wedge (S.i = Idle) \wedge (\ (Child.i \neq NULL) \vee$
  $(ph.i \neq ph.P.i)\ )\ ]\ )\ ]$

Fig. 8. Miscellaneous Rule.

R14 : $\text{Faulty}(i) \rightarrow\ S.i = Erroneous\ ;\ P.i = NULL.$

---

4 Correctness of the algorithm

For the lack of space, we give only the proof sketch.

LS is the region where the following conditions are true: (i) the satisfied privileges are R0, R1, R2, R3, R4, R5, or/and R6; (ii) there exist neither any cycle nor any illegal tree; (iii) there exists no *Erroneous* processor.
We will prove that (i) all computations reach *LS*, (ii) *LS* is closed, and (iii) in *LS*, our algorithm constructs a BFS spanning tree.

To prove the correctness of our algorithm, we use the *convergent stair* [GM91] method. We exhibit a finite sequence of regions A_0, A_1, ... , A_n where A_0 is the set of system states, A_n is *LS* and for all i < n A_{i+1} is an A_i-attractor. First, we establish that computations are infinite.

Theorem 1. *In any system state, at least one processor holds a privilege.*

We consider three global configurations: (i) there is a conflict in the network (the processor in conflict holds R12 or R13 privilege, its parent holds R11 privilege, or one of its children holds R8 privilege), (ii) there is an *Erroneous* processor inside a tree (a processor inside its subtree holds R8, R9, or R11 privilege), and (iii) there is no *Erroneous* processor inside a tree and there is no conflict. In

configuration (iii) , there are several situations: (iiia) the legal tree has an *Idle* processor whose phase value differs from that of its parent, (this processor hold the R3, R4, or R14 privilege); (iiib) the legal tree has a *Power* processor (one of its neighbors holds R2, R7, R10, or R14 privilege; or it holds the R0, R1, R5, or R6 privilege); (iiic) the legal tree has a *Working* leaf (this leaf holds the R6 or R0 privilege); and (iiid) any configuration other than (iiia), (iiib) and (iiic). In situation (iiid), the legal tree has a *Working* processor whose children have the *Idle* status. Either this processor holds R1 or R5 privilege, or one of its children holds R14 privilege.

Theorem 2. *Let A0 be the set of system states.*
A1 $\equiv$ {*No processor satisfies the Faulty predicate*} *is an A0-attractor.*

As long as a processor satisfies the Faulty predicate, it holds R14 privilege. Eventually, (by the fairness assumption) all processors will be in a correct state (*A1* will be reached). No move puts a processor in a faulty state.

Theorem 3. *A2* $\equiv$ *A1* $\cap$ {*If a processor is inside an illegal tree, then it has Erroneous status*} *is an A1-attractor.*

R14 is the only rule that can create a new illegal root. Therefore, in *A1*, the creation of an illegal tree cannot happen. But, the illegal tree may gain new processors by R2 or R13 moves, as long as they have *Power* processors. However, by induction, we prove that all processors inside illegal tree will take *Erroneous* status. As long as an illegal root does not have *Erroneous* status, this processor holds R7 privilege. By fairness, all illegal roots will acquire the *Erroneous* status. Assume that all processors inside an illegal tree and n-1-away from their root have *Erroneous* status. They cannot have a new child (illegal trees cannot gain new processors that are n-away from their root). All the illegal processors n-away have *Erroneous* parent. By fairness scheduling of R8, these processors will take *Erroneous* status.

Theorem 4. *A3* $\equiv$ *A2* $\cap$ {*There exists no illegal tree*} *is an A2-attractor.*

In *A2*, no more processor will join an illegal tree and the illegal trees will eventually destroy themselves (by fairness scheduling of rule R9).

Theorem 5. *A4* $\equiv$ *A3* $\cap$ {*The Erroneous-free cycles do not have an Influential processor*} *is an A3-attractor.*

The difficulty is that the cycles (path without a root) may gain new processors. Indeed, the processors (called *influential*) that have *Power* status or that will possibly take *Power* status (by a series of R3 and R4 moves), may gain children. The cycles may contain influential processors.
Only R0 and R1 moves allow new processors to become influential. These processors are not inside any cycle. Nevertheless, a cycle may gain influential processors: a subtree of the legal tree may join a cycle by executing R13. After such a move, the cycle has at least one *Erroneous* processor which is the processor

that executed R13 move.
Therefore, the cycles without an *Erroneous* processor cannot gain an influential processor without gaining an *Erroneous* processor. By the fairness assumption, the influential processors inside an Erroneous-free cycle will get *Power* status, and will eventually change their status (thus, they will be no more influential).

Theorem 6. *$A5 \equiv A4 \cap \{$There exists no irregular and no influential processor$\}$ is an $A4$-attractor.*

An *irregular* processor is inside a cycle, or it is inside the legal tree, but it does not have color equal to the *r_color*, or one of its ancestors had joined the legal tree by a R13 move. Thus, a regular processor was initially inside the legal tree or it joined the legal tree by a R2 move. A regular processor has color equal to *r_color*. Only R0 and R1 allow new processors, that are *regular*, to become influential. As mentioned before, the regular and influential processors may become irregular, when one of their ancestors executes a R13 move. After verifying the EndFirstPhase predicate, a regular processor does not quit the legal tree by a R13 move. Indeed, when the EndFirstPhase predicate is verified by a regular processor, all its neighbors have color equal to *r_color*. Either its neighbors are influential (they cannot change their color while they are influential), or they change their color (they have become irregular but they cannot become influential till they are irregular).
After the first R0 move, before satisfying the predicate EndFirstPhase, the regular processors may only execute a R13 move. Therefore, when a regular processor executes a R13 move, its subtree contains only *Erroneous* processors (children). As regular processors have only regular ancestors (after the first R0 move), the regular and influential processors cannot quit the legal tree by a R13 move.
Thus, after the first R0 move, there will be no new irregular and influential processor. As irregular and influential processors have an *Erroneous* ancestor, they will eventually take *Erroneous* status (by fairness scheduling of the rules R8 and R11).
We need to prove that every computation contains a R0 move in order to prove that *A5* is an *A4*-attractor.

Theorem 7. *A computation cannot have a finite number of R0 moves.*

Assume that a computation, $\mathcal{C}$, with a finite number of R0 moves exists. Along $\mathcal{C}$, at some point, no R0 move will be executed, and the legal tree will never have an *Erroneous* processor (otherwise it would eventually destroy itself and a R0 move would be executed). Thus, after the last R0 move, no processor inside the legal tree executes a R13 move. Therefore, the network will never have new irregular and influential processors, and eventually, the irregular and influential processors will be removed. Then, no processor may execute a R13 move. Otherwise, the legal tree would get an *Erroneous* processor (now, *Power* processors are only inside the legal tree). At that point, the processors inside the legal tree may only execute R0, R1, R3, R4, R5, and R6 moves. The processors inside a cycle may execute R5, R6, R8, R9, R11, and R12 moves. The detached

processors may execute R2 and R10 moves. Now, each processor may execute at most five moves, between two consecutive R1 moves. As $\mathcal{C}$ is an infinite computation, (theorem 1), there is an infinite number of R1 moves after the last R0 move. Between two consecutive R1 moves, the legal tree gains at least one processor. Since the network is finite, the legal tree cannot gain processors forever. Therefore, there is a finite number of R1 moves after the last R0 move. We proved the contradiction.

Theorem 8. *$A6 \equiv A5 \cap \{$There exists no cycle$\}$ is an A5-attractor.*

In *A5*, the cycles cannot gain a processor by a R2 or R3 move because they do not have a *Power* processor. No new cycle may be created because *Power* processors and their ancestors are regular (they have the color equal to *r_color*). We know that all computations contain an infinite number of R0 moves. Thus, all r's neighbors execute an infinite number of R2 moves. Otherwise, one of r's neighbors, say i, would keep its color forever, after a R0 move, i and r would not have the same color, and then r would never satisfy the predicate EndFirstPhase. Thus, it would not be able to execute any move.
Between two R2 moves, a processor must satisfy the predicate EndFirstPhase: after a R2 move, a processor executes a R4 move. Then, it needs to satisfy the predicate EndFirstPhase in order to execute its next move (R5 or R6).
Since we have established that all r's neighbors execute an infinite number of R2 moves, we prove that if a computation contains an infinite number of R2 moves executed by a processor, then the computation contains also an infinite number of R2 moves executed by its neighbors. Therefore, by induction on the distance between the processors and r, we can establish that each processor executes an infinite number of R2 moves in all computation. Only the detached processors may execute a R2 move. Thus, no processor stays forever inside a cycle.

Theorem 9. *$LS \equiv A6 \cap \{$No processor has the Erroneous status$\}$ is an A6-attractor.*

Clearly, LS is closed. If the legal tree has an *Erroneous* processor, it will destroy itself and a R0 move will eventually be executed. After the R0 move, only the detached processors have *Erroneous* status. These processors hold R10 privilege till they have *Erroneous* status. By fairness scheduling of the rule R10, *LS* will be reached. Once *LS* is reached, no processor will get the *Erroneous* status.

Let s_0 (s_1) be the global state where all processors are detached and are 0-colored (1-colored). At the end of a tree construction, more processors will have the color equal to *r_color*. Thus, in any computation in *LS*, s_0 or s_1 will be eventually reached. Then, the construction will complete and be well-formed: as no processor has initially the color equal to *r_color*; a processor ends its first phase after all its neighbors have joined the legal tree (and taken the color equal to *r_color*). The r's neighbors will join the legal tree during the first phase. By induction, we can conclude that the processors at a distance of k from r will join the legal tree during the kth phase.

We proved that (i) LS is an $A0$-attractor; and (ii) in LS, our algorithm constructs a BFS spanning tree.

References

[AB97] Yehuda Afek and Anat Bremler. Self-stabilizing unidirectional network algorithms by power-supply. In *Sixth ACM-SIAM Symposium on Discrete Algorithms (SODA)*, San Francisco, January 1997. to appear in Chicago Journal of Theoretical Computer Science.

[AKY90] Yehuda Afek, Shay Kutten, and Moti Yung. Memory-efficient self-stabilizing protocols for general networks. In *WDAG'90*, volume 486, pages 15–28. LNCS, Springer-Verlag, September 1990.

[Ang80] D. Angluin. Local and global properties in networks of processors. In *Proceedings of the 11th Annual ACM Symposium on Theory of Computing*, pages 82–93, 1980.

[AO94] Baruch Awerbuch and Rafail Ostrovsky. Memory-efficient and self-stabilizing network reset. In *Symposium on Principles of Distributed Computing*, pages 254–263, 1994.

[BLB95] Franck Butelle, Christian Lavault, and Marc Bui. A uniform self-stabilizing minimum diameter spanning tree algorithm. In *WDAG'95*, volume 972, pages 257–272. LNCS, Springer-Verlag, September 1995.

[CYH91] Nian-Shing Chen, Hwey-Pyng Yu, and Shing-Tsaan Huang. A self-stabilizing algorithm for constructing spanning trees. *Information Processing Letters*, 39:147–151, 1991.

[Dij74] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the A.C.M.*, 17(11):643–644, 1974.

[DIM93] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Self-stabilization of dynamic systems assuring only read/write atomicy. *Distributed Computing*, 7:3–16, 1993.

[Dol93] Shlomi Dolev. Optimal time self stabilization in dynamic systems. In *WDAG'93*, volume 725, pages 160–173. LNCS, Springer-Verlag, September 1993.

[GM91] M.G. Gouda and N. Multari. Stabilizing communication protocols. *IEEE Transactions on Computing*, 40(4):448–458, 1991.

[HC92] Shing-Tsan Huang and Nian-Shing Chen. A self-stabilizing algorithm for constructing breadth-first trees. *Information Processing Letters*, 41:109–117, 1992.

[HC93] Shing-Tsan Huang and Nian-Shing Chen. Self-stabilizing depth-first token circulation on networks. *Distributed Computing*, 7:61–66, 1993.

[IL94] Gene Itkis and Leonid Levin. Fast and lean self-stabilizing asynchronous protocols. In *35th Symposium on Foundations of Computer Science*, pages 226–239, Santa Fe, New Nexico, December 1994. IEEE Computer Sciety Press.

[JB95] Colette Johnen and Joffroy Beauquier. Space-efficient distributed self-stabilizing depth-first token circulation. In *Second Workshop on Self-Stabilizing Systems*, May 1995.

[Sch93] Marco Schneider. Self-stabilization. *ACM Computing Surveys*, 25(1):45–67, 1993.

[SS92] Sumit Sur and Pradip K. Srimani. A self-stabilizing distributed algorithm to construct bfs spanning trees on a symetric graph. *Parallel Processing Letters*, 2(2/3):171–179, 1992.

[TH94] Ming-Shin Tsai and Shing-Tsaan Huang. A self-stabilizing algorithm for the shortest paths problem with a fully distributed demon. *Parallel Processing Letters*, 4(1):65–72, 1994.

This article was processed using the LaTeX 2_ε macro package with SIROCCO class