Information Processing Letters 39 (1991) 147–151
North-Holland 16 August 1991

A self-stabilizing algorithm for constructing spanning trees *

Nian-Shing Chen
Department of Information Management, National Sun Yat-Sen University, Taiwan, ROC

Hwey-Pyng Yu and Shing-Tsaan Huang
Institute of Computer Science, National Tsing Hua University, Hsinchu, Taiwan, ROC

Communicated by S.G. Akl
Received 23 April 1990
Revised 20 April 1991

Abstract

Chen, N.-S., H.-P. Yu and S.-T. Huang, A self-stabilizing algorithm for constructing spanning trees, Information Processing Letters 39 (1991) 147–151.

A self-stabilizing algorithm is proposed for constructing spanning trees for connected graphs. Because of the self-stabilizing property, the algorithm can handle error recovery automatically.

Keywords: Analysis of algorithms, combinatorial problems, design of algorithms, fault tolerance, spanning tree, self-stabilizing

1. Introduction

This paper proposes a self-stabilizing algorithm for constructing spanning trees in distributed systems. The term *self-stabilizing* was originally introduced by Dijkstra [3] to distinguish any system having the property that starting at any, possibly illegitimate state, the system is guaranteed to converge to a legitimate state in finite time, and once in a legitimate state it will remain so forever. Such a property is very desirable for any distributed system because after any unexpected perturbation, the system eventually recovers and returns to a legitimate state without any outside intervention. Other works addressing the self-stabilizing problem can be found in [1,2,5–7].

Consider a connected graph $G(V, E)$, in which V is a set of nodes and E is a set of edges. Such a graph is used to model a distributed system with n processors, $n = |V|$, in which each node represents a processor. In the graph, directly connected nodes are called each other's *neighbors*. We use the terms *node* and *processor* interchangeably in the paper. Our goal is to design a self-stabilizing algorithm that maintains a spanning tree for the graph with each node knowing its level in the tree.

The proposed algorithm has several rules for each processor. Each rule has two parts: the *antecedent* (condition) part and the *move* part. The antecedent part is defined as a boolean function of the processor's own state and the states of its neighbors; when the antecedent part of any rule is true, we say that the processor has the *privilege*. A processor that has the privilege may then make the corresponding *move*, which brings the processor into a new state that is a function of its old state and the states of its neighbors.

* This research was supported by the National Science Council of the Republic of China under the contract NSC79-0408-E007-03.

We assume the existence of a central daemon as suggested in [3]. In some instances, many processors may have the privilege at the same time. However, we assume that only one processor makes the move at a time, and the privilege for the next move depends on the states resulting from the previous move. This implies that the rules are *atomic*; the processors cannot evaluate their privilege at a time and then make the move later with moves of others in between. After completion of a move, the daemon arbitrarily selects a new privilege, viz. a new processor having the privilege to apply the rule.

In the proposed algorithm, each processor except a specific one which we call the *root* maintains two local variables *level* and *parent*. The values of *level* range over $\{1, \ldots, n\}$ and the *parent* of a processor points to one of its neighbors. The system is said to be in a *legitimate state* if the parent pointers constitute a spanning tree of G rooted at the root, and each processor except the root has a level equal to the level of its parent plus one; otherwise, it is in an *illegitimate state*. The level of a processor is the distance from the processor to the root along the tree edges.

The proposed algorithm is self-stabilizing because regardless of the initial state (possibly illegitimate) and regardless of the privilege selected each time for the next move, the algorithm always builds a spanning tree after a finite number of moves for the system with each processor knowing its level in the tree. Once the system is in a legitimate state, the algorithm terminates, i.e., no processor has the privilege. And after that, any unexpected perturbation would bring the system again into an illegitimate state; the algorithm is reactivated automatically, however, and the system is guaranteed to be in a legitimate state after the algorithm terminates again.

The proposed algorithm meets the following requirements:

(i) In each illegitimate state, at least one processor has the privilege.

(ii) When the system is in a legitimate state, no processor has the privilege, i.e., the algorithm terminates.

(iii) Regardless of the initial state and regardless of the privilege selected each time for the next move, the system is guaranteed to reach a legitimate state after a finite number of moves.

Note that, requirement (i) implies that if the system state is changed from a legitimate state to an illegitimate state by any unexpected perturbation, the algorithm is reactivated automatically.

Our work is different from other related works in the sense that we are interested in developing fault-tolerant graph algorithms, instead of token-based algorithms. Spanning trees are the most fundamental structures for many graph algorithms. This is the motivation of our work.

Proving correctness of self-stabilizing algorithms is nontrivial [4,6]. Kessels [6] reported that a straightforward way to show a system self-stabilizing is: first prove that the system can always make a move as long as the system is not stabilized and then give a bounded function whose value decreases for each move. However, it may not be easy to find such a function. As will be seen, our proof follows Kessels's approach; we use such a bounded function in the proof.

2. The proposed algorithm

As described earlier, we use a connected graph $G(V, E)$ to model a distributed system. A specific node r in V is selected as the root. The algorithm is designed to construct from G a spanning tree rooted at r with each node knowing its level in the tree.

Each node i other than the root maintains the following two local variables:

$L(i)$: level of i,

$P(i)$: parent of i,

where $l \leqslant L(i) \leqslant n$ and $P(i)$ is a neighbor of node i. The initial values of those variables are unpredictable but within their domain. The root node r has a constant level $L(r) = 0$ and no parent variable.

Let p denote the parent of node i, i.e., $p = P(i)$. Our rules eventually construct from G a tree rooted at node r. In the tree, we have $L(r) = 0$ and for any other node i, we require that $L(i) = L(p) + 1$.

That is, when the system reaches a legitimate state, the following predicate is true:

$$\text{GST} \equiv (\forall i, p\colon i \neq r \wedge p = P(i)\colon\ L(i) = L(p) + 1).$$

Hence, a node whose level is not equal to the level of its parent plus one should make a move by changing its level. Rule (R0) is designed to handle this situation. Because $|V| = n$ and $L(r) = 0$, in a legitimate state the level of any node cannot be greater than $n - 1$. Hence, an error state at a node i is represented by the value $L(i) = n$. If the parent of a node is in an error state, the node must be in an error state too. Rule (R1) is designed to deal with such an error propagation. Once a node is in an error state, it should try to recover itself from the error state. Rule (R2) is designed for the recovery. Note that a processor has the privilege if the antecedent part of any rule is true.

(R0) $L(i) \neq n \wedge L(i) \neq L(p) + 1 \wedge L(p) \neq n$
$\quad\rightarrow L(i) := L(p) + 1.$

(R1) $L(i) \neq n \wedge L(p) = n \rightarrow L(i) := n.$

(R2) Let k be some neighbor of i,
$\quad L(i) = n \wedge L(k) < n - 1$
$\quad\rightarrow L(i) := L(k) + 1;\ P(i) := k.$

Example. In Fig. 1, a connected graph $G(V, E)$ with $V = \{a, b, c, d, e\}$ is given. Node c is the root. Numbers in the figure are levels, and the arrows represent the parent pointers. For example, in the initial configuration, $P(d) = b$. The rules by which a node has the privilege are shown in the figure; the underlined rule is the one that is selected for the next move.

After six moves we get a final configuration, a spanning tree rooted at node c with each node knowing its level in the tree. A different moving sequence may produce a tree that is different from the one we have presented. The readers can convince themselves by working out this example with different moving sequences.

3. Verification

We now show that the algorithm meets the requirements (i), (ii) and (iii). According to the definition of GST, the reader can easily verify that the algorithm meets requirement (ii). In the following, we will prove that the algorithm satisfies requirements (i) and (iii).

The main ideas in the reasoning are as follows. First, we show that before GST is true, the al-

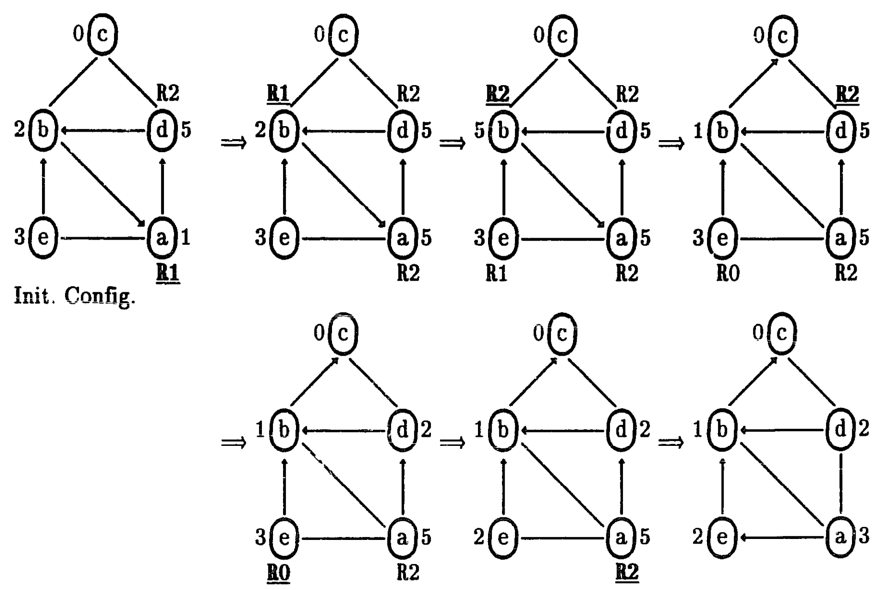

Fig. 1. An example to show how the rules work.

gorithm does not terminate; i.e., the design meets requirement (i). Second, we define an evaluation function F over the configurations of the system. Whenever a node in the system makes a move, the function has its value decreased. Hence, we can eventually have GST = true. Since GST = true implies the configuration is a tree rooted at node r with each node knowing its level, we have the system eventually reached a legitimate state. That is, the design meets requirement (iii).

A parent pointer $i \to p$ is called a *Well-Formed* (WF) pointer if $L(i) \neq n$, $L(p) \neq n$ and $L(i) = L(p) + 1$. For any configuration, if we consider the WF pointers only, we can uniquely partition the nodes of the graph into several trees, i.e., the graph is a spanning forest. For each tree in the forest, we define a WF set to be the set of nodes in the tree. For example, in the initial configuration of Fig. 1, pointers $e \to b$ and $b \to a$ are WF pointers, and $a \to d$ and $d \to b$ are non-WF pointers. The nodes of the graph are partitioned into three trees defined by the WF pointers; so we have three WF sets $\{a, b, e\}$, $\{c\}$ and $\{d\}$. Note that in any configuration the WF sets are mutually disjoint. Further, any node i, $L(i) = n$, constitutes a singleton WF set.

Over any WF set S, there is a directed spanning tree. Let i, $i \in S$, be the node with the minimum level in the set. Then, the spanning tree is rooted at i. By the rules, all nodes on the spanning tree other than the root have no privilege. To emphasize the root and its level, we use $S_i^{L(i)}$ to denote the WF set rooted at i. For example, in the initial configuration of Fig. 1, we have $S_c^0 = \{c\}$, $S_a^1 = \{a, b, e\}$, and $S_d^5 = \{d\}$.

Lemma 1. *Before GST is true, the algorithm does not terminate.*

Proof. Before GST is true, there must exist some WF set $S_i^{L(i)}$, $i \neq r$. Two possible cases need to be considered.

Case 1: $L(i) \neq n$ for some WF set $S_i^{L(i)}$, $i \neq r$. Node i has the privilege by either (R0) or (R1) because either $L(p) \neq n$ or $L(p) = n$, where p is the parent of i.

Case 2: $L(i) = n$ for every WF set $S_i^{L(i)}$, $i \neq r$. Since G is connected, there exists at least one edge between a node k in S_r^0, and some node i, $i \neq r$ and $S_i^{L(i)} = S_i^n = \{i\}$. Because $L(r) = 0$ and $|S_r^0| < n$, we have $L(k) < n - 1$. Hence, node i can apply (R2) to make the move. This completes the proof. □

Fix a configuration, and let t_k, $0 \leqslant k \leqslant n$, be the number of WF sets $S_i^{L(i)}$ such that $L(i) = k$. Define the value of F for this configuration to be

$$(t_0, t_1, \ldots, t_n), \quad 0 \leqslant t_i < n.$$

The comparison of the values of F is lexicographic: $(a_0, a_1, \ldots) > (b_0, b_1, \ldots)$ if there exists some k, such that $a_i = b_i$, $0 \leqslant i < k$, and $a_k > b_k$.

It can be easily seen that F is a bounded function with maximum value $(1, n-1, 0, \ldots, 0)$ and minimum value $(1, 0, \ldots, 0)$. For any possible initial state, we have $L(r) = 0$, $r \in S_r^0$, and no

Step 0:	$S_c^0 = \{c\}, S_a^1 = \{a, b, e\}, S_d^5 = \{d\}$	$F = (1, 1, 0, 0, 0, 1)$
Step 1:	$S_c^0 = \{c\}, S_b^2 = \{b, e\}, S_a^5 = \{a\}, S_d^5 = \{d\}$	$F = (1, 0, 1, 0, 0, 2)$
Step 2:	$S_c^0 = \{c\}, S_e^3 = \{e\}, S_a^5 = \{a\}, S_b^5 = \{b\}, S_d^5 = \{d\}$	$F = (1, 0, 0, 1, 0, 3)$
Step 3:	$S_c^0 = \{c, b\}, S_e^3 = \{e\}, S_a^5 = \{a\}, S_d^5 = \{d\}$	$F = (1, 0, 0, 1, 0, 2)$
Step 4:	$S_c^0 = \{c, b, d\}, S_e^3 = \{e\}, S_a^5 = \{a\}$	$F = (1, 0, 0, 1, 0, 1)$
Step 5:	$S_c^0 = \{c, b, d, e\}, S_a^5 = \{a\}$	$F = (1, 0, 0, 0, 0, 1)$
Step 6:	$S_c^0 = \{c, b, d, e, a\}$	$F = (1, 0, 0, 0, 0, 0)$

Fig. 2.

other nodes can have their level equal to 0; hence, we always have $t_0 = 1$.

An example that follows Fig. 1 is shown in Fig. 2 to demonstrate the changing of F when the system makes moves. Step 0 is for the initial configuration, the other steps correspond to the six moves described in Fig. 1.

From this example, we can see how the system moves toward the configuration at which $F = (1, 0, \ldots, 0)$. In other words, the system converges to a single WF set S_r^0. And, each time when a node makes the move, the value of function F decreases.

Lemma 2. *F monotonically decreases each time when rule* (R0), (R1) *or* (R2) *is applied.*

Proof. Each application of rule (R0), (R1) or (R2) is by the root node i of some WF set $S_i^{L(i)}$. If node i applies (R0) to make the move, then after the application, node i appends to some existing WF set, and the rest of the set $S_i^{L(i)}$, if not empty, splits into WF set(s) $S_k^{L(k)}$ with $L(k) = L(i) + 1$. Hence, $t_{L(i)}$ decreases by one, and $t_{L(i)+1}$ may increase.

If node i applies (R1) to make the move, then after the application, node i constitutes a new singleton WF set $S_i^n = \{i\}$, and the rest of set $S_i^{L(i)}$, if not empty, splits into WF set(s) $S_k^{L(k)}$ with $L(k) = L(i) + 1$. Hence, $t_{L(i)}$ decreases by one, t_n increases by one and $t_{L(i)+1}$ may increase.

If node i applies (R2) to make the move, then the singleton set S_i^n combines with some existing WF set. Hence, t_n decreases by one.

From the above discussion, according to lexicographical order, F monotonically decreases each time when rule (R0), (R1) or (R2) is applied. □

Theorem 3. *Eventually, the system reaches a legitimate state.*

Proof. Since the initial value of F is finite, and the smallest possible value for F is $(1, 0, \ldots, 0)$, by Lemma 2 the rules can only be applied a finite number of times. Hence, by Lemma 1, eventually GST is true.

When GST is true, it is obvious that the configuration is a tree of G rooted at node r with each node knowing its level in the tree. □

4. Remarks

Rules of a self-stabilizing algorithm are said to be *noninterfering*, if once a processor has the privilege, it remains so unless the corresponding move is made [1]. If the rules have the noninterfering property, they can be nonatomic, i.e., the processors can concurrently evaluate different parts of the antecedents of their rules at different times and then make the move later. Hence, the noninterfering property is very desirable; but, it may be difficult to design rules with such a property.

Although the proposed rules are interfering, due to the monotonic decreasing property of the bounded function, we have a strong feeling that they work correctly even if they are not atomic. We are investigating the issue.

Acknowledgment

The authors want to thank the referees for their suggestions for revising the paper.

References

[1] G.M. Brown, M.G. Gouda and C.L. Wu, Token systems that self-stabilize, *IEEE Trans. Comput.* **38** (6) (1989) 845-852.

[2] J.E. Burns and J. Pachl, Uniform self-stabilizing rings, *ACM Trans. Programming Language Systems* **11** (2) (1989) 330-344.

[3] E.W. Dijkstra, Self-stabilizing systems in spite of distributed control, *Comm. ACM* **17** (11) (1974) 643-644.

[4] E.W. Dijkstra, A belated proof of self-stabilization, *Distributed Comput.* **1** (1) (1986) 5-6.

[5] S.-T. Huang, An O(n) self-stabilizing algorithm for uniform rings, Submitted for publication.

[6] J.L.W. Kessels, An exercise in proving self-stabilizing with a variant function, *Inform. Process. Lett.* **29** (1988) 39-42.

[7] H.S.M. Kruijer, Self-stabilization (in spite of distributed control) in tree-structured systems, *Inform. Process. Lett.* **8** (1979) 91-95.