Information Processing Letters 43 (1992) 77–81
North-Holland

24 August 1992

A self-stabilizing algorithm for maximal matching *

Su-Chu Hsu and Shing-Tsaan Huang
Institute of Computer Science, National Tsing Hua University, Hsinchu 30043, Taiwan, ROC

Communicated by K. Ikeda
Received 20 May 1991
Revised 9 April 1992

Abstract

Hsu, S.-C. and S.-T. Huang, A self-stabilizing algorithm for maximal matching, Information Processing Letters 43 (1992) 77–81.

We present a self-stabilizing algorithm for finding a maximal matching in distributed networks. Due to its self-stabilizing property, the algorithm can automatically detect and recover from the faults caused by unexpected perturbations on local variables maintained on each node of the system. A variant function is provided to prove the correctness of the algorithm and to analyze its time complexity.

Keywords: Distributed systems, fault-tolerance, maximal matching, self-stabilization, variant function

1. Introduction

This paper presents a self-stabilizing algorithm for finding a maximal matching in distributed networks. In any distributed system faults caused by unexpected perturbations on local variables are possible. The system should have the ability to detect the faults and recover from them. The term *self-stabilizing*, originally introduced by Dijkstra [4], is used to distinguish distributed systems with the property that the systems can automatically recover from any illegitimate state and reach a legitimate state in a finite time. The most attractive feature for a self-stabilizing system is that each node can, simply by its local view, detect and recover from the transient faults. Such a property is very desirable for distributed systems with fault-tolerance. The concept of self-stabilization was regarded by Lamport [12] as Dijkstra's most brilliant contribution in distributed systems and a milestone in the work of fault tolerance. Recent works addressing the self-stabilizing problems of distributed systems can be found in [2,4–11].

Consider an undirected graph $G(V, E)$ where V is a node set and E is an edge set. A *matching* of $G(V, E)$ is a set of edges M, $M \subseteq E$, in which no two edges connect to a common node. A matching M is *maximal* if it is not properly contained in any other matching. Note that M is a maximal matching does not necessarily imply that M has more edges than any other matching.

Finding a maximal matching sequentially can be carried out simply by inspecting the edges of

Correspondence to: Professor S.-T. Huang, Institute of Computer Science, National Tsing Hua University, Hsinchu 30043, Taiwan, ROC. Email: sthuang@cs.nthu.edu.tw.

* This research is supported by the National Science Council of the Republic of China under the contract NSC80-0408-E007-04.

the graph one by one [3]. An inspected edge is added to the matching if it is not adjacent to any of the edges that are already there. Unfortunately, a straightforward self-stabilizing implementation of this approach can hardly be found in distributed systems. In the proposed self-stabilizing algorithm, nodes can mutually select and then decide whether they get matched or not. Due to the self-stabilizing property of our proposed algorithm, each node can locally detect and recover from the transient faults caused by unexpected perturbations on local variables. For a self-stabilizing algorithm, proving its correctness and analyzing its time complexity are nontrivial. Following the technique reported by Kessels [11], a variant function is provided to prove the correctness of the proposed algorithm. We further use the variant function to analyze its time complexity. On a general graph with n vertices, for the worst case, the upper bound $O(n^3)$ is obtained. To find the proper variant function is not trivial and constitutes the main contribution of this paper.

The rest of this paper is organized as follows. In Section 2, the proposed self-stabilizing algorithm is presented. In Section 3, the correctness and the time complexity of the algorithm are analyzed. Some remarks are discussed in Section 4.

2. The proposed algorithm

Consider a distributed system with the structure $G(V, E)$, where $|V| = n$. Assume that each node i knows $N(i)$, the set of its adjacent nodes (neighbors) and $N(i)$ is always correct. We let each node i maintain a pointer. The pointer points to one of i's neighbors which i selects to match. If i's pointer points to null, it means i does not select anyone. We use the notation $i \rightarrow j$ to denote that the pointer of i points to j, and the notation $i \rightarrow null$ to mean that the pointer of i points to null. We also use $i \Leftrightarrow j$ to represent $i \rightarrow j$ and $j \rightarrow i$; i.e., i and j mutually select and get matched. Due to unexpected perturbations, the pointers may be affected and vary.

We use S.i to denote the state of node i. If $i \rightarrow j$, then there are three possible states defined as follows:

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

If S.i = *waiting*, it means i has selected j and waits for j to select it. If S.i = *matched*, it means i has gotten matched. If S.i = *chaining*, it means i has selected j but j has selected another node.

If $i \rightarrow null$, then there are two possible states defined as follows:

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

If S.i = *dead*, it means i has no chance to get matched, because all its neighbors have gotten matched. If S.i = *free*, it means that it still may have chances to get matched, even though i does not select anyone.

Note that if S.i = *free*, then ($\forall j$: $j \in N(i)$: S.j $\neq$ *dead*). We will use this fact to prove Lemma 1 in Section 3. If $i \rightarrow null$, the S.i may be *free* or *dead*. This will be used to prove Lemma 2 in Section 3. A global state of the system is defined as a collection of all states of the nodes in the system.

By the definition of maximal matching, when the system reaches a maximal matching, each node's state must be either *matched* or *dead*. Thus the system is said to be in a legitimate state (i.e., stabilized) if each node's state is either *matched* or *dead*. Therefore, the following predicate GMM is introduced. When GMM is true, it means the system reaches a legitimate state.

$$\text{GMM} \equiv (\forall i\text{:: S}.i = \textit{matched} \vee \text{S}.i = \textit{dead})$$

We propose a self-stabilizing algorithm such that a system starting at any illegitimate state is guaranteed to converge to a legitimate state in a finite time and remains so thereafter. The algorithm is identically stored and executed in each node i. The self-stabilizing algorithm is expressed by a set of rules. The rules are expressed as:

"*condition* $\Rightarrow$ *a corresponding move*".

Conditions of each node are defined to be boolean functions of its own pointer and the pointers of its neighbors. When the condition of a node is true, the node may make the *corresponding move*. Any node for which the condition is true is said to have *privilege*. In some instances, many nodes may have the privileges at the same time. However, we require to decide which nodes to make the moves at a time, and the next moves depend on the result from the previous moves. This implies that the rules are atomic. We assume there exists a *central demon* as introduced in [4] which is used to select one of the nodes with the privileges. The node enjoying the selected privilege will then make its move by modifying its pointer.

The self-stabilizing algorithm for maximal matching

(R1) $(i \rightarrow \mathit{null}) \wedge (\exists j\colon j \in N(i)\colon j \rightarrow i)$
$\Rightarrow$ Let $i \rightarrow j$

(R2) $(i \rightarrow \mathit{null}) \wedge (\forall k\colon k \in N(i)\colon \neg(k \rightarrow i))$
$\wedge (\exists j\colon j \in N(i)\colon j \rightarrow \mathit{null})$
$\Rightarrow$ Let $i \rightarrow j$

(R3) $(i \rightarrow j) \wedge (j \rightarrow k) \wedge (k \neq i)$
$\Rightarrow$ Let $i \rightarrow \mathit{null}$

3. Verification

The correctness of our algorithm requires the fulfillment of the following requirements:

(i) If the system reaches a legitimate state, no nodes can make further moves.

(ii) If the system is in any illegitimate state, there exists at least one node which can make a move.

(iii) Regardless of the initial state and regardless of which node is selected to make a move by the central demon, the system is guaranteed to reach a legitimate state after a finite number of moves.

According to the definition of GMM, it is obvious that when GMM is true, no rules can be applied. Thus, our design meets requirement (i). The following Lemma 1 proves that our design also satisfies requirement (ii).

Lemma 1. *If* GMM *is false, there exists at least one node which can make a move*; *i.e., there exists some rule to be applied.*

Proof. If GMM is false, then it means that there must exist some node whose state is neither *matched* nor *dead*; i.e., GMM is false $\Rightarrow$ $(\exists i\colon\colon \neg(S.i = \mathit{matched} \vee S.i = \mathit{dead}))$. The following cases need to be considered:

(1) $S.i = \mathit{chaining}$: It is clear that (R3) can be applied by node i.

(2) $S.i = \mathit{waiting}$: There exists j such that $i \rightarrow j$ and $j \rightarrow \mathit{null}$. Hence, obviously (R1) can be applied by node j.

(3) $S.i = \mathit{free}$: By the definition of *free*, there must exist some neighbor j of i whose state is not *matched*. Since $S.j$ cannot be *dead* as mentioned earlier, $S.j$ must be *free, waiting* or *chaining*. The following describes the three cases:

(i) $S.j = \mathit{free}$: It is clear that (R2) can be applied by either i or j.

(ii) $S.j = \mathit{waiting}$: Similar to (2), (R1) can be applied by some k ($j \rightarrow k$ and $k \rightarrow \mathit{null}$).

(iii) $S.j = \mathit{chaining}$: Similar to (1), (R3) can be applied by j.

From (1), (2) and (3), we can get that if GMM is false there exists some rule to be applied. □

In order to prove our design meets requirement (iii), we use a verification method based on a variant function which can be found in [2,6,8–11]. The basic concepts of the method are: (1) to give a variant function whose value is bounded; (2) to prove the variant function monotonically decreases or increases when nodes make moves.

Let m, d, w, f and c denote the total number of nodes whose state are *matched, dead, waiting, free* and *chaining* respectively. We define the variant function F as:

$$F \equiv (m + d, w, f, c).$$

The comparison of the values of F is by lexicographical order. Each global state of the system corresponds to one value of F.

By Lemma 1, if the system is not in a legitimate state, then the nodes of the system can always make moves. This causes F to vary. By the definition of GMM, the value of F corresponding to the legitimate state is $(n, 0, 0, 0)$. Clearly, it is the upper bound of F. Thus, if we can prove that F monotonically increases for each move, our design will meet requirement (iii). The following lemma will show this idea.

Lemma 2. *F monotonically increases each time when rule* (R1), (R2) *or* (R3) *is applied.*

Proof. (1) If (R1) is applied then there will be a pair of nodes which can get *matched*. In other words,

$$(\mathrm{S}.i = \mathit{free}) \wedge (\mathrm{S}.j = \mathit{waiting} \wedge j \to i)$$
$$(\text{after } i \text{ applies (R1)})$$
$$\Rightarrow (\mathrm{S}.i = \mathit{matched}) \wedge (\mathrm{S}.j = \mathit{matched}).$$

Because the states of i and j are changed to *matched*, the states of some neighbors of i and/or j may be changed from *free* to *dead*. Furthermore, it should be clear that no node can have its state changed from *matched* or *dead* to any other state no matter which rule is applied.

Therefore from the above, after (R1) is applied, no matter how states of nodes will be affected, we know at least that m increases by 2 and d may also increase. It follows that F increases.

(2) If (R2) is applied by node i, then it means

$$(i \to \mathit{null}) \wedge (\forall k\colon k \in N(i)\colon \neg(k \to i))$$
$$\wedge (\exists j\colon j \in N(i)\colon j \to \mathit{null})$$

before the application of (R2). It can be easily verified that after i applies (R2), only S.i is changed from *free* to *waiting*. In other words,

$$(\mathrm{S}.i = \mathit{free})$$
$$(\text{after } i \text{ applies (R2)})$$
$$\Rightarrow (\mathrm{S}.i = \mathit{waiting}).$$

Thus, after (R2) is applied, m, d and c are unchanged, f decreases by 1 and w increases by 1. It follows that F increases.

(3) If (R3) is applied by node i, then $i \to \mathit{null}$ after the application of (R3). Only the following two cases are possible:

(i) $(\mathrm{S}.i = \mathit{chaining}) \wedge (\forall k\colon k \in N(i)\colon \neg(k \to i))$
(after i applies (R3))
$\Rightarrow (\mathrm{S}.i = \mathit{free} \wedge \mathrm{S}.i = \mathit{dead})$

(ii) $(\mathrm{S}.i = \mathit{chaining}) \wedge (k \to i)$
/* Note that S.k = *chaining*. */
(after i applies (R3))
$\Rightarrow (\mathrm{S}.i = \mathit{free}) \wedge (\mathrm{S}.k = \mathit{waiting})$

In case (i), it is clear that although c decreases by 1, either f or d should increase by 1. Hence, F increases. In case (ii), although c decreases at least by 2, f should increase by 1 and w should increase at least by 1. Thus, F increases.

By (1), (2) and (3), we have that F monotonically increases each time when rule (R1), (R2) or (R3) is applied. □

The following theorem will show the upper bound of the time complexity of the algorithm and Lemma 3 is given to support this theorem. The proof of Lemma 3 can be found in [1].

Lemma 3. *The number of the non-negative integer solutions* (x_1, x_2, x_3, x_4) *for the equation* $x_1 + x_2 + x_3 + x_4 = n$ *is*

$$\binom{n+3}{3} = \frac{(n+1)^*(n+2)^*(n+3)}{6}.$$

Theorem 4. *Regardless of any initial state of the system, the system will converge to a legitimate state within* $O(n^3)$ *moves.*

Proof. (1) By Lemma 1 and Lemma 2, regardless of any initial value of F, F will converge to its upper bounded value in a finite number of moves.

(2) Regardless of which global state of the system, 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. Suppose that in the worst case, the initial value of F is $(0, 0, 0, n)$, one would like to know how many moves it may need to transfer F from $(0, 0, 0, n)$ to $(n, 0, 0, 0)$ in the worst case. Such a problem can be reduced to the problem de-

scribed in Lemma 3 with $x_1 = m + d$, $x_2 = w$, $x_3 = f$ and $x_4 = c$. Thus, the maximal total number of moves for the system to converge to a legitimate state is

$$\frac{(n+1)^*(n+2)^*(n+3)}{6} - 1 = \mathrm{O}(n^3).$$

By (1) and (2), the theorem is proved. □

4. Conclusions

In this paper, we propose a self-stabilizing algorithm for finding a maximal matching in distributed networks. A variant function is provided to prove the correctness of the algorithm and to analyze the time complexity. In the worst case, the upper bound $\mathrm{O}(n^3)$ moves is obtained. However,, there are two problems associated with the proposed algorithm: (1) how to analyze and reduce the upper bound of the time complexity; (2) how to relax the requirement of (R1) and (R2) 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. Our future work will be devoted to solving such problems.

Because of the randomness of the network topology, the initial state of the system and the selection of moving sequences, it is difficult to analyze the average number of moves for the proposed algorithm. However, the derivation of the average time complexity of self-stabilizing algorithms is an important problem.

References

[1] R.C. Bose and B. Manvel, *Introduction to Combinatorial Theory* (Wiley, New York, 1984) 48.

[2] N.-S. Chen, F.-P. Yu and S.-T. Huang, A self-stabilizing algorithm for constructing spanning trees, *Inform. Process. Lett.* **39** (1991) 147–151.

[3] N. Deo, *Graph Theory with Applications to Engineering and Computer Science* (Prentice-Hall, Englewood Cliffs, NJ, 1974).

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

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

[6] M.G. Gouda and T. Herman, Stabilizing unison, *Inform. Process. Lett.* **35** (1990) 171–175.

[7] T. Herman, Probabilistic self-stabilization, *Inform. Process. Lett.* **35** (1990) 63–67.

[8] S.-T. Huang, Self-stabilizing leader election in uniform rings, *ACM Trans. Programming Language Systems*, to appear.

[9] S.-T. Huang and N.-S. Chen, A self-stabilizing algorithm for constructing breadth-first trees, *Inform. Process. Lett.* **41** (1992) 109–117.

[10] S.-C. Hsu and S.-T. Huang, A generalized self-stabilizing protocol for centrality problems on tree networks, in: *Proc. of National Computer Symp.*, Taoyuen, Taiwan, ROC, pp. 59–64.

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

[12] L. Lamport, Solved problems, unsolved problems and non-problems in concurrency, in: *Proc. 3rd ACM Symp. on Principles of Distributed Computing* (1984) 1–11.