IEICE TRANS. INF. & SYST., VOL.E93–D, NO.10 OCTOBER 2010

LETTER

The Time Complexity of Hsu and Huang's Self-Stabilizing Maximal Matching Algorithm

Masahiro KIMOTO[†a)], ***Nonmember***, **Tatsuhiro TSUCHIYA**[†], ***Member***, ***and*** **Tohru KIKUNO**[†], ***Fellow***

SUMMARY The exact time complexity of Hsu and Huan's self-stabilizing maximal matching algorithm is provided. It is $\frac{1}{2}n^2 + n - 2$ if the number of nodes n is even and $\frac{1}{2}n^2 + n - \frac{5}{2}$ if n is odd.

key words: *self-stabilization, maximal matching, time complexity, stabilization time, distributed computing*

1. Introduction

The idea of self-stabilization was introduced by Edsger W. Dijkstra in 1974 [1]. Self-stabilizing algorithms enable systems to be started in an arbitrary state and still converge to a desired behavior.

In this letter, we discuss the time complexity of the self-stabilizing algorithm proposed by Hsu and Huang in [2], which finds a maximal matching in a network. This algorithm is the first self-stabilizing maximal matching algorithm and has been regularly cited in the literature.

Because of its technical importance, the time complexity of this particular algorithm has been well studied. In [2], Hsu and Huang show that it is bounded by $O(n^3)$, where n is the number of nodes. In [3], Tel provides an almost tight upper bound, which is $\frac{1}{2}n^2+2n+1$ if n is even and $\frac{1}{2}n^2+n-\frac{1}{2}$ if n is odd. In [4] Tel gives a more concise proof for the $O(n^2)$ bound than [3]. In [5] Hedetniemi, Jacobs and Srimani provide an upper bound of $2m + n$, where m is the number of edges. This gives a better bound than the one in [3] only if the network is sparse. In this letter, we provide the exact time complexity of the Hsu-Huan algorithm.

2. The Hsu-Huang Algorithm

We consider a distributed system consisting of n ($\geq$ 2) nodes. The topology of the system is modeled as an undirected graph. Let $N(p)$ denote the set of a node p's adjacent nodes (neighbors). Each node p has a pointer. The pointer either points to one of p's neighbors which p selects to match or has a null value. The notation $p \rightarrow q$ denotes that p's pointer points to $q \in N(p)$, the notation $p \rightarrow null$ denotes that p's pointer has a null value, and the notation $p \Leftrightarrow q$ denotes that $p \rightarrow q \wedge p \rightarrow q$.

Each node p is in either one of the following five states.

1. If $\exists q \in N(p) : (p \rightarrow q) \wedge (q \rightarrow null)$, then p is *waiting*.
2. If $\exists q \in N(p) : p \Leftrightarrow q$, then p is *matched*.
3. If $\exists q \in N(p), \exists r \in N(q) : (p \rightarrow q) \wedge (q \rightarrow r) \wedge (r \neq p)$, then p is *chaining*.
4. If $(p \rightarrow null) \wedge (\forall q \in N(p) :$ q is matched$)$, then p is *dead*.
5. If $(p \rightarrow null) \wedge (\exists q \in N(p) :$ q is not matched$)$, then p is *free*.

A maximal matching is found if and only if every node is either matched or dead.

The Hsu-Huang algorithm at each node p is given by the following three rules.

(R1)

$$(p \rightarrow null) \wedge (\exists q \in N(p) : q \rightarrow p)$$
$$\Rightarrow \text{Let } p \rightarrow q$$

(R2)

$$(p \rightarrow null) \wedge (\forall r \in N(p) : \neg(r \rightarrow p))$$
$$\wedge (\exists q \in N(p) : q \rightarrow null)$$
$$\Rightarrow \text{Let } p \rightarrow q$$

(R3)

$$(p \rightarrow q) \wedge (q \rightarrow r) \wedge (r \neq p)$$
$$\Rightarrow \text{Let } p \rightarrow null$$

where each rule is of form *guard* $\Rightarrow$ *action*. Each rule is executed atomically and no two processes can execute a rule at a time.

A *configuration* of the system is a collection of the pointers of all nodes. The execution of a rule by a node causes a transition from a configuration to the next configuration. We define a *run* as a sequence of configurations, $s_1, s_2, \cdots, s_l$ such that the transition from s_i to s_{i+1} is possible for any $i \leq l-1$. In [2], it is proven that any run of the algorithm is finite and every node is either matched or dead in the last configuration of any maximal run, meaning that the system always converges a configuration where a maximal matching is obtained. The time complexity of the algorithm is the maximum number of *steps* (that is, rule executions) required to find a maximal matching. Thus we have:

$$\text{(time complexity)} = \text{(the length of the longest run)} - 1$$

Manuscript received May 18, 2010.
Manuscript revised June 30, 2010.

[†]The authors are with Osaka University, Suita-shi, 565–0871 Japan.
a) E-mail: m-kimoto@ist.osaka-u.ac.jp

DOI: 10.1587/transinf.E93.D.2850

3. Upper Bound on Run Lengths

Our derivation of the upper bound on the time complexity follows the basic line of [3]. In [3], as in much of the self-stabilization literature, the time complexity is analyzed using the *variant function* technique. A *variant function* is a function over configurations, whose value is monotonically decreases (in our context) when nodes execute a rule of the algorithm.

Our variant function is a tuple (F, G), where F and G are functions that map a configuration to a non-negative integer as follows:

$$F \equiv \begin{cases} \left\lceil \frac{c+f+w}{2} \right\rceil & \text{(even } n\text{)} \\ \left\lfloor \frac{c+f+w}{2} \right\rfloor & \text{(odd } n\text{)} \end{cases}$$
$$G \equiv 2c + f$$

where c, f, and w are the number of chaining, free and waiting nodes, respectively. For odd n, the variant function is identical to that of [3]. The modification made is that a different expression of G is used for even n. As stated later, this subtle modification is critical in obtaining the exact time complexity.

The variable function is evaluated in the lexicographical order; i.e., F is evaluated first and then G. Below we show that this function indeed monotonically decreases when a rule is executed. As in [3], an important observation is that $c + f + w$ never increases because, by the design of the rules, matched or dead nodes remain matched or dead. Thus it suffices to see that either F or G is decreased by an execution of a rule. In the following description, p, q, and r refer to p, q, and r in the rule definition described in Sect. 2.

(1) Execution of Rule R1

Rule R1 is enabled only when p is free and q is waiting. When it is executed, p and q become matched. Dead or matched nodes do not change their state. Hence the rule execution decreases $c + f + w$ by at least 2, thus decreasing F by at least 1.

(2) Execution of Rule R2

Rule R2 is enabled only when p is free and causes p to become waiting. Because no node is waiting for p ($\forall r \in N(p) : \neg(r \rightarrow p)$), no waiting node becomes chaining nor free. Except p, all free nodes remain free. Hence the execution of the rule decreases G by 1.

(3) Execution of Rule R3

This rule is enabled only when p is chaining and causes p to be free or dead by setting p's pointer to *null*. No node becomes chaining. Also no waiting node becomes free, as p is the only node that makes its pointer *null*. Hence the execution of R3 decreases G by at least 1.

In summary, the execution of any of the three rules either (1) decreases F or (2) decreases G but not increase F. This leads to the following observation:

Observation 1. *Any run $s_1 s_2 \cdots s_l$ is a concatenation of runs $\sigma_1, \cdots, \sigma_m$ such that: (1) all configurations in σ_i have the same F value; (2) if configurations s, s' occur in σ_i and σ_{i+1} respectively, then $F(s) > F(s')$; (3) if configurations s, s' consecutively occur in σ_i, then $G(s) > G(s')$.*

This is schematically represented as follows:

$$\begin{array}{cccccc} & \overbrace{\cdots \quad s_{j-2} \qquad s_{j-1}}^{\sigma_i} & & \overbrace{s_j \qquad s_{j+1} \quad \cdots}^{\sigma_{i+1}} & \\ \cdots & \cdots = F(s_{j-2}) = F(s_{j-1}) & > & F(s_j) = F(s_{j+1}) = \cdots & \cdots \\ & \cdots > G(s_{j-2}) > G(s_{j-1}) & & G(s_j) > G(s_{j+1}) > \cdots & \end{array}$$

Another observation used in obtaining the upper bound is as follows:

Observation 2. *Because a waiting node waits for a free node, $w \geq 1$ implies $f \geq 1$. Hence $G \geq 1$ if $c + f + w \geq 1$; $G = 0$ if $c + f + w = 0$.*

Our derivation of the upper bound refines the one by Tel in [3] in three points.

- The analysis of the configurations where $F = \lfloor \frac{n}{2} \rfloor$ is refined (Lemma 1). This reduces the upper bound by 4 if n is even and by 2 if n is odd.
- The new variable function allows to reduce the upper bound by $n - 2$ for the case of even n (Lemma 2).
- The analysis of the configurations where $F = 0$ is refined (Lemma 3). This reduces the upper bound by 1 for the case of even n.

As a result, the new bound is smaller than that of [3] by $n+3$ if n is even and by 2 if n is odd.

Lemma 1. *If a run $s_1 s_2 \cdots s_l$ satisfies $F(s_1) = \cdots = F(s_l) = \lfloor \frac{n}{2} \rfloor$, then the length l of the run is at most $2n - 2$.*

Proof. From Observation 2, $G(s_i) \geq 1$ for any s_i. If $G(s_1) \leq 2n - 2$, then the lemma trivially follows. Thus in the following of the proof, we assume that $2n - 1 \leq G(s_1) \leq 2n$ and proceed as follows: We first show that under this assumption, there is always a "cycle" of chaining nodes in s_1. Then we show that $l \leq 2n - 2$ holds in the two cases: (1) none of the nodes consisting of the cycle executes a rule in the run; and (2) some node in the cycle executes a rule.

Because of the assumption of $2n - 1 \leq G(s_1) \leq 2n$, either $c = n - 1 \wedge f = 1 \wedge w = 0$ or $c = n \wedge f = w = 0$ holds in s_1. Hence in s_1 every chaining node has a pointer pointing to another chaining node. (Note that even if $f = 1$, a chaining node cannot point to that free node, since if a node p has a pointer to a free node, then, by definition, p is a waiting node.)

In s_1, therefore, there is at least one cycle of chaining nodes; that is, there is a sequence of nodes $p_1, p_2, \cdots, p_{len}$ such that $p_i \rightarrow p_{i+1}$ for all $i, 1 \leq i \leq len - 1$ and $p_{len} \rightarrow p_1$. By the definition of a chaining node, the cycle contains at least three nodes; that is, $len \geq 3$.

If none of the nodes consisting of the cycle executes a rule in the run, $G(s_l) \geq 6$, because $len \geq 3$ implies $c \geq 3$. In that case, since $2n \geq G(s_1) > G(s_2) > \cdots > G(s_l) \geq 6$, the run length l is at most $2n - 5$.

Now consider the case where some node in the cycle executes a rule in this run. Note that only R3 can be executed by this node. Let p be the first node that executes the rule in the cycle and s_i be the configuration in which this rule execution occurs. Then $G(s_i) - 3 \geq G(s_{i+1})$, because p becomes free and the node that points to p in the cycle becomes waiting, resulting in a decrease in c by at least 2 and an increase in f by 1. Since $2n \geq G(s_1) > G(s_2) > \cdots > G(s_l) \geq 1$, the run length l is at most $2n - 2$. □

Lemmas 2, 3, 4 apply to the case of even n.

Lemma 2. *When n is even, if a run $s_1 s_2 \cdots s_l$ satisfies $F(s_1) = \cdots = F(s_l) = x > 0$, then the length l of the run is at most $4x$.*

Proof. $G(s_1) \leq 4x$, since c is at most $2x$. From Observations 1 and 2, $4x \geq G(s_1) > G(s_2) > \cdots > G(s_l) \geq 1$; thus the lemma follows. □

Lemma 3. *When n is even, if a run $s_1 s_2 \cdots s_l$ satisfies $F(s_1) = \cdots = F(s_l) = 0$, then the length l of the run is 1.*

Proof. When n is even, if $F = 0$, then $c + f + w = 0$ and $2f + w = 0$. Hence the run contains exactly one configuration in which every node is matched or dead. □

Lemma 4. *When n is even, the length of any run is at most:*

$$\frac{1}{2}n^2 + n - 1$$

Proof. As there are n nodes, $0 \leq F \leq \frac{n}{2}$ and $0 < \frac{n}{2}$. From Observation 1 and Lemmas 1, 2, 3, the upper bound on the run length is derived as follows:

$$\begin{aligned}&(2n-2) + \sum_{x=1}^{\frac{n}{2}-1} 4x + 1\\ &= \frac{1}{2}n^2 + n - 1\end{aligned}$$

□

Lemmas 5, 6, 7 apply to the case of odd n.

Lemma 5. *When n is odd, if a run $s_1 s_2 \cdots s_l$ satisfies $F(s_1) = \cdots = F(s_l) = x > 0$, then the length l of the run is at most $4x + 2$.*

Proof. $G(s_1) \leq 4x + 2$, since c is at most $2x + 1$. From Observations 1 and 2, $4x + 2 \geq G(s_1) > G(s_2) > \cdots > G(s_l) \geq 1$; thus the lemma follows. □

Lemma 6. *When n is odd, if a run $s_1 s_2 \cdots s_l$ satisfies $F(s_1) = \cdots = F(s_l) = 0$, then the length l of the run is at most 2.*

Proof. At each configuration s_i in the run, either $c+f+w = 1$ or $c + f + w = 0$, because $F(s_i) = 0$.

If $c + f + w = 1$, then $2c + f = 2$, since neither $c = w = 0 \wedge f = 1$ nor $c = f = 0 \wedge w = 1$ is possible: This is because a node can be free or waiting only if at least one of its neighbors is neither matched nor dead. If $c + f + w = 0$, then $2c + f = 0$. As a result, $G(s_i) = 2$ (if $c + f + w = 1$) or $G(s_i) = 0$ (if $c + f + w = 0$) for any s_i in the run, thus the run length is at most 2. □

Lemma 7. *When n is odd, the length of any run is at most:*

$$\frac{1}{2}n^2 + n - \frac{3}{2}$$

Proof. As there are n nodes, $0 \leq F \leq \lfloor\frac{n}{2}\rfloor$ and $0 < \lfloor\frac{n}{2}\rfloor$. From Observation 1 and Lemmas 1, 5, 6, the upper bound on the run length is derived as follows:

$$\begin{aligned}&(2n-2) + \sum_{x=1}^{\lfloor\frac{n}{2}\rfloor-1} (4x + 2) + 2\\ &= \frac{1}{2}n^2 + n - \frac{3}{2}\end{aligned}$$

□

4. The Exact Time Complexity

In this section we provide the exact time complexity, by showing an algorithm execution whose run length exactly matches the upper bound obtained in the previous section. This example of execution is identical to that shown by Tel in [3]; however the run length is analyzed only for the case of even n. Here we provide the exact run length for the case of odd n, generalizing his result to any n. Before presenting our result, we first describe this execution in order to clearly show how the result is derived.

Suppose that the system consists of $n \geq 3$ nodes $p_1, p_2, \cdots, p_n$ and that the topology of the system is a complete graph. Also suppose that initially $p_1 \to p_2$, $p_2 \to p_3, \cdots, p_{n-1} \to p_n$, $p_n \to p_1$.

1. R3 is executed by $n - 1$ nodes $p_1, p_2, \cdots, p_{n-1}$. As a result, all the $n - 1$ nodes become free and p_n becomes waiting for p_1.
2. R2 is executed by $n - 2$ nodes $p_2, p_3, \cdots, p_{n-1}$ to point p_1.
3. R1 is executed by p_1 to match p_n. As a result, p_1 and p_n become matched and the other $n - 2$ nodes become chaining.

Phases 1–3 result in $n - 2$ chaining nodes and two matched nodes.

Phases 4–6 starts with $n - i$ matched nodes and i (≥ 2) chaining nodes pointing to a matched node. Initially i is $n - 2$.

4. R3 is executed by the i chaining nodes. As a result, all the i nodes become free.

5. Let p be any one of the i free nodes. The free nodes other than p execute R2 to point p. The $i-1$ steps cause these free nodes to become waiting.
6. R1 is executed by one of the waiting node, say q, to match p. As a result, p and q become matched and the other $i-2$ nodes become chaining. Phases 4–6 is repeated with i replaced with $i-2$ until at most one chaining node remains.

As a result, all nodes become matched if n is even, whereas a single chaining node remains if n is odd. In the latter case, Phase 7 is performed.

7. R3 is executed by the chaining node, causing the node to become dead.

The number of steps of the above execution is expressed as follows: For even n:

$$2n-2+\sum_{x=1}^{\frac{n}{2}-1}(2x+(2x-1)+1)$$
$$=\frac{1}{2}n^2+n-2$$

For odd n:

$$2n-2+\sum_{x=1}^{\lfloor\frac{n}{2}\rfloor-1}(2(x+1)+2x+1)+1$$
$$=\frac{1}{2}n^2+n-\frac{5}{2}$$

For the case $n = 2$, we can consider the following scenario. Starting with two free nodes, the execution of R2 by each of the nodes leads to a final configuration where both are matched. The number of steps involved in this example is two, which coincides with the above expression.

Theorem 1. *The exact time complexity of the algorithm is expressed as follows.*

$$\frac{1}{2}n^2+n-2 \quad (\text{even } n)$$
$$\frac{1}{2}n^2+n-\frac{5}{2} \quad (\text{odd } n)$$

Proof. By Lemma 4 and Lemma 7, these expressions represent the upper bound on the time complexity. The above examples of algorithm execution show that these expressions also represent the lower bound. □

5. Conclusion

We analyzed the time complexity of Hsu and Huang's self-stabilizing maximal matching algorithm [2]. Refining the result by Tel [3], we provided the exact time complexity.

References

[1] E.W. Dijkstra, "Self-stabilizing systems in spite of distributed control," Commun. ACM, vol.17, no.11, pp.643–644, 1974.

[2] S.C. Hsu and S.T. Huang, "A self-stabilizing algorithm for maximal matching," Inf. Process. Lett., vol.43, no.2, pp.77–81, 1992.

[3] G. Tel, "Maximal matching stabilizes in quadratic time," Inf. Process. Lett., vol.49, no.6, pp.271–272, 1994.

[4] G. Tel, Introduction to Distributed Algorithms, Cambridge University Press, New York, NY, USA, 2001.

[5] S.T. Hedetniemi, D.P. Jacobs, and P.K. Srimani, "Maximal matching stabilizes in time $o(m)$," Inf. Process. Lett., vol.80, no.5, pp.221–223, 2002.