An Anonymous Self-Stabilizing Algorithm For 1-Maximal Matching in Trees

Wayne Goddard, Stephen T. Hedetniemi
Department of Computer Science, Clemson University
{goddard,hedet}@cs.clemson.edu

Zhengnan Shi (SPEAKER)
Department of Computer and Information Sciences
State University of New York Institute of Technology
shic@sunyit.edu

KeyWords: self-stabilizing, matching, anonymous, trees, rings

Abstract. *We present an anonymous self-stabilizing algorithm for finding a 1-maximal matching in trees, and rings of length not divisible by 3. We show that the algorithm converges in $O(n^4)$ moves under an arbitrary central daemon.*

1 Introduction

Networks of computers are used to solve a wide variety of problems. Most services for a network involve maintaining a global predicate over the entire network by using local information at each participating node. To this end, several paradigms for distributed algorithms have been studied. Among these are the (fault-tolerant) self-stabilizing algorithms introduced by Dijkstra [2]. The self-stabilizing property is that a system can start in any configuration and yet is guaranteed to converge to a desired configuration in finite time and remain so thereafter [3, 4, 10]. In this paper we provide a self-stabilizing algorithm which produces an improved maximal matching.

1.1 Self-stabilization

In the self-stabilizing algorithmic paradigm, each node maintains and changes its own local variables based on the current values of its variables and those of its *neighbors*. The values of the local variables at a node determine its *local state*. The *global state* of a network is the union of all the *local states*.

When a node changes its local state, it is said to make a *move*. The algorithm is presented as a set of rules. Each rule is of the form "*if* $p(i)$ *then* $q(i)$" where i is a node, $p(i)$ is a boolean predicate and $q(i)$ a move. A node is said to be *privileged* if $p(i)$ is true. If a node becomes privileged, it may execute the corresponding move $q(i)$.

When no node is privileged, we say that the system is in a *stable state*. For us, an algorithm is *self-stabilizing* if: from any initial global state it reaches a legitimate global stable state after a finite number of moves. (This is sometimes called a silent self-stabilizing algorithm.)

[0]Second and third authors supported in part by NSF grant #0218495

We do not make use of IDs: the network is *anonymous*. However, the ability of anonymous, self-stabilizing algorithms to solve problems is relatively weak, and thus our algorithm works only for limited networks. Like most matching papers (e.g. [7]), we assume nodes have pointers and a node can determine whether its neighbor points to it. This can be achieved by every two adjacent nodes sharing a private register (called the *link-register* model).

As is standard in deterministic anonymous algorithms, we assume a *central daemon* [2, 7], which at each time-step selects one of the privileged nodes to make a move: thus no two nodes move at the same time. We also assume *coarse atomicity*: a node can read all its neighbors' variables and update its own variables in one move. Because of the indeterminism, it is impossible to simulate the daemon selection process—self-stabilizing algorithms usually converge rapidly when simulated with a random daemon, but mathematical techniques are needed to show convergence under all circumstances.

Anonymous self-stabilizing algorithms for some graph problems have been given in [5, 6, 7, 8, 9].

1.2 Our result

We model a distributed network as a connected, undirected graph $G = (V, E)$. For a node $i \in V$, $N(i)$, its *open neighborhood*, is the set of nodes to which i is adjacent. A *matching* is a subset M of edges such that each node in V is incident to at most one edge in M. A matching is *maximal* if $\forall e \in E - M$ it holds that $M \cup \{e\}$ is not a matching.

An anonymous self-stabilizing algorithm for maximal matching in any connected network was presented by Hsu and Huang [7]. Though the algorithm uses pointers and appears to use IDs, a node only needs to know whether a neighbor points to (a) it, (b) nothing, or (c) another node. A maximal matching algorithm for a model with fewer assumptions (but using randomness to generate IDs) was presented by Chattopadhyay, Ligham and Seyffarth [1].

A maximal matching M is *1-maximal* if it satisfies the following property: $\forall e \in M$, no matching can be constructed by removing e from M and adding two edges to $M - \{e\}$. Our result is:

Theorem 1 *We present an anonymous, self-stabilizing algorithm that:*
(a) for any network produces a 1-maximal matching if it stabilizes; and
(b) is guaranteed to stabilize if the underlying graph is a tree or a ring with length not divisible by 3.

We show that the performance guarantee of a 1-maximal matching algorithm is better than that of a maximal matching algorithm. We also show that for rings of length a multiple of 3, no deterministic, anonymous, 1-maximal matching algorithm is possible.

2 Description of Algorithm

Our algorithm builds on the maximal matching algorithm of Hsu and Huang [7]. In their algorithm, every node maintains a pointer. When the system stabilizes, the pointers are in pairs, and an edge is in the matching precisely when its two ends point to each other. In this case, each matched node points to its *spouse*.

To obtain a 1-maximal matching algorithm, we need to provide a mechanism for a *2-for-1 exchange* where one edge of the matching is replaced by two. This is achieved by a *hand-shaking* mechanism: the two matched nodes signal to each other the availability of alternate spouses, lock in commitments from these alternates, and then point to their new spouses.

2.1 Rough version

We present first an overview of the algorithm in informal language. Of necessity, certain concepts will not be defined precisely (or will be inaccurate) until we present the actual rules.

In the algorithm, each node i maintains two variables: (i) a pointer, and (ii) a local variable x_i. The pointer records the neighbor to which i points (its intended spouse). We write $i \to j$ if i's pointer is set to j, and $i \to \emptyset$ if i's pointer is set to null.

The variable x_i records the current state of the node i; this is used to indicate progress towards a 2-for-1 exchange. There are four states, denoted 0, 1, 2 and 3. Informally one may describe the intent of each state as follows:

- State 0 is a quiet state. Any unmatched node, as well as a matched node that cannot achieve a 2-for-1 exchange, should be in this state.
- State 1 should mean that a node is matched and has an alternate neighbor available (i.e., an unmatched, uncommitted neighbor).
- State 2 should mean that a node is matched and its spouse also has an alternate neighbor available. This state is a signal that beckons the available neighbor to commit to the node by pointing to it.
- State 3 should mean that a node is matched and has a committed neighbor. This is a signal to its spouse to break the match if it too has a committed neighbor.

2.2 The actual rules

We start with the intention of the rules, in informal language. We group the rules into three pairs.

The first pair of rules are designed for a *node that is not in a match*. These two rules echo the rules of the Hsu–Huang maximal matching algorithm:

1. BackUp: if you are pointing to a node that does not want you to point to it (it is pointing elsewhere and not beckoning), then set your pointer to null.
2. Point: if your pointer is null and there is a neighbor that wants you to point to it (it is pointing to you, to null, or is beckoning a commitment), then point to that neighbor.

The second pair of rules are designed for a *node that is matched* in order for it to implement the hand-shaking procedure. The state of a node is meant as a signal to its spouse: it indicates progress in informing nodes and obtaining commitments from nodes to execute a 2-for-1 exchange.

3. StateIncrease: if you see the potential to expand the number of edges in the matching—because of your spouse's state or a suitable neighbor—then go to the next higher state.
4. Abort: if your state does not correctly represent the situation—because of changes by your neighbors or because your data was not initialized—then reset your state.

The final pair of rules are designed to enable a 2-for-1 exchange to occur: the two spouses each marry a committed neighbor.

5. BreakMatch: if your spouse has agreed to the 2-for-1 exchange, and you have a committed neighbor, then point to the neighbor (and reset your state).
6. Remarry: if your spouse has already changed, then point to a committed neighbor.

The exact rules of the algorithm are shown in Figure 1.

(1) BackUp: **if** $i \to j \ \wedge \ j \nrightarrow i \ \wedge \ j \nrightarrow \emptyset \ \wedge \ x_j < 2$
then $x_i = 0 \ \wedge \ i \to \emptyset$

(2) Point: **if** $i \to \emptyset$
$\wedge \ \exists$ incoming, beckoning or open $k \in N(i)$
then $x_i = 0 \ \wedge \ i \to k$
(the selection order for k is incoming neighbors in increasing state, then beckoning neighbors, then open neighbors)

(3) StateIncrease: **if** $x_i < 3$
$\wedge \ i$ is in a proper match with j with $x_i \leq x_j$
$\wedge$ (*either* $\exists$ incoming $k \in N(i)$, *or* $x_i < 2$ and $\exists$ open $k \in N(i)$)
then $x_i{+}{+}$

(4) Abort: **if** $x_i > 0 \ \wedge$ not eligible for Rule Remarry $\wedge$
$\wedge$ (*either* i is not in a proper match, *or* $\nexists$ incoming nor open $k \in N(i)$,
or $x_i = 3$ and $\nexists$ incoming $k \in N(i)$)
then set $x_i = 2$ if $x_i = 3$, i's spouse is in State 1 and i has an incoming neighbor; and set $x_i = 0$ otherwise.

(5) BreakMatch: **if** i is in a proper match with a spouse in State 3
$\wedge \ \exists$ incoming $k \in N(i)$ that is not in State 3
then $x_i = 0 \ \wedge \ i \to k$

(6) Remarry: **if** $x_i = 3 \ \wedge \ i \nrightarrow \emptyset \ \wedge \ i$ is not matched
$\wedge \ \exists$ incoming $k \in N(i)$
then $x_i = 0 \ \wedge \ i \to k$

Abbreviations:
i is in *proper match* with j if $i \to j$, $j \to i$ and $|x_i - x_j| \leq 1$.
$k \in N(i)$ is *beckoning* to i if $i \nrightarrow k$ and $x_k = 2$.
$k \in N(i)$ is *incoming* if $i \nrightarrow k$ and $k \to i$.
$k \in N(i)$ is *open* if $i \nrightarrow k$ and $k \to \emptyset$.

Figure 1: The 1-Maximal Matching Algorithm (the rules for node i)

3 Analysis of Algorithm

We show first that the algorithm is *correct* for any network: every stable state has the desired global property. Then we show that in the special networks of trees and cycle of length not a multiple of 3, the algorithm is guaranteed to reach a stable state after polynomially many moves.

3.1 Correctness

In this section we show that every stable state has the desired global property. To show this correctness, we show that at convergence there is no node in state 2 or 3. Then we show that this means the pointers represent a 1-maximal matching.

For a value y of the state variable x, we use the notation $\mathcal{S}(y)$ to represent the set of nodes i for which $x_i = y$.

Lemma 3.1 *In a stable state produced by this algorithm,* $\mathcal{S}(2) = \mathcal{S}(3) = \emptyset$.

PROOF. Suppose upon stabilization there is a node $u \in \mathcal{S}(2)$. Because u is not privileged for Rule Abort, it must be in a proper match.

Let v be u's spouse: we know $x_v \geq 1$. If $x_v = 1$, then v is privileged for either Rule StateIncrease or Abort. So assume $x_v \geq 2$. Then the only way u is privileged for neither of those rules, is if it has an open neighbor, say z. However, then node z is privileged for Rule Point, a contradiction. Thus in a stable state, $\mathcal{S}(2)$ is empty.

Suppose there is a node $w \in \mathcal{S}(3)$. Since $\mathcal{S}(2) = \emptyset$, and w cannot execute Rule Abort, w is in a proper match with a spouse in State 3. Node w cannot have an incoming neighbor in State 3, since that neighbor would be privileged for Rule Abort. Thus w is privileged either for Rule Abort or for Rule BreakMatch, depending on whether it has an incoming neighbor. This is a contradiction. $\Diamond$

Lemma 3.2 *In any network, if the algorithm stabilizes then it produces a 1-maximal matching given by nodes pointing to each other.*

Proof. By Lemma 3.1, in a stable state the network contains nodes only in States 0 and 1. By Rule BackUp and Rule Point, if $i \to j$ then $j \to i$. Thus, an edge of the matching is given by spouses pointing to each other.

Suppose the resultant matching is not maximal. Then there is a pair of adjacent nodes whose connecting edge can be added: but they are privileged for Rule Point.

Similarly, if the resultant matching is not 1-maximal, then there is a path of four nodes a, b, c, d such that neither a nor d is in the matching but the edge bc is. Since every node is in State 0 or 1, by Rule BackUp both a and d point to $\emptyset$. Thus both b and c are privileged for Rule StateIncrease, a contradiction. $\Diamond$

3.2 Convergence and complexity analysis

We need to prove that, in some networks, the algorithm always reaches a stable state after finitely many moves no matter what the central daemon does. There are two parts to showing convergence.

The first part is to introduce the concept of a *progress move*. We show that the number of progress moves is bounded. A useful concept here is the idea of a node being *semi-matched*. The set of matched nodes does not monotonically increase—we need the ability to break matches and remarry. Nevertheless, we show that once a node becomes semi-matched, it remains semi-matched (and we later show that the node will eventually become matched in the special networks). The first part holds true in any network.

For the second part, we show that, for the special networks we are interested in, the daemon cannot cause the network to move indefinitely without a progress move occurring. The idea here is that if the process goes on forever, there must be nodes which are *repeatedly starting the 2-for-1 exchange process* and then aborting it. Since there is no overall progress, this abortion is caused by an available neighbor committing itself to another 2-for-1 exchange. That exchange in turn is aborted because of another neighbor committing elsewhere, and so on. This process can go on forever if there is a cycle back to the first exchange. However, we show that an infinite process is impossible in trees, and in rings whose length is not divisible by 3.

3.2.1 Semi-matched nodes and progress moves

Lemma 3.3 *The number of matches never decreases.*

Proof. The only move (Rule BreakMatch) that deletes an existing match replaces it with one. $\Diamond$

We say that a node is *dirty* if it has not moved, otherwise it is *clean*. We say that a node is *semi-matched* if: (a) it points to null and is pointed to by a node that is not in state 3; or (b) it is in state 3 and is pointed to (but not matched).

Let T denote the set of nodes that are matched or semi-matched. The first lemma shows that one a node actively becomes semi-matched, then it remains so. The easy proof is omitted.

Lemma 3.4 *A node u in T remains in T except possibly when u is dirty and in State 3.*

We define a *progress move* as either: (1) a move of a dirty node in State 3; (2) a move that creates a new match; or (3) a move that changes a null pointer to point to an open neighbor.

Lemma 3.5 *There are at most $O(n^2)$ progress moves.*

PROOF SKETCH. Clearly there are at most n Type 1 progress moves. By considering all the possibilities, one can argue that every Type 2 move increases the set T and so there are $O(n)$ Type 2 moves. A Type 3 move freezes that node and any subsequent move by an open neighbor creates a match; thus there are at most $O(n^2)$ Type 3 moves. $\Diamond$

3.2.2 Between progress moves in special networks

In many networks, however, the daemon can cause ensure no progress move occurs and yet the algorithm does not terminate. So in some networks, including trees and cycles of the appropriate length, we now argue that the adversarial daemon cannot go forever without causing progress.

We therefore focus on the maximum number of moves that can be made between two progress moves. During such a period, the set of edges in the matching remains unchanged. Indeed, the nodes that are matched can execute only Rule StateIncrease or Abort.

A node that is not matched cannot change its pointer to an incoming or open neighbor, since both are progress moves. We define a u-to-v *flip* as a move by node u that changes its pointer from null to point to a beckoning neighbor v. We define a *flop* as a node executing Rule BackUp. We define a *flopflip* as a flop followed by a flip.

Lemma 3.6 *Consider a period during which there is no progress move. Assume a node u flopflips to v three times. Then between the first flop and the final flip, some node x adjacent to v's spouse w flopflips to a node other than w.*

PROOF SKETCH. During this time w must oscillate between state 0 and state at least 1, meaning that it changes from not having an open neighbor to having an open neighbor and back again. That neighbor is the x we seek. $\Diamond$

By iterating the above argument we obtain:

Corollary 3.1 *Consider a period during which there is no progress move. If a node u flopflips to v k times, then one can identify a sequence of $k-2$ flopflips $f_1, \ldots, f_{k-2}$ as follows. The flopflip f_i is executed by some neighbor x_i of v's spouse ($x_i \neq v$), and points to some node other than v or u, and the flop part of f_i occurs between the ith flop by u and the $(i+2)$nd u-to-v flip.*

We define the *cause* of a flopflip as follows. We say the first two u-to-v flopflips are *self-caused*. For a later u-to-v flopflip, we assign the *cause* as another flopflip by some node x, as in the above lemma. Associated with each cause, other than self-causes, is a directed *cause path* from u to x.

Now, the flopflip by x in turn, has a cause. We can therefore follow the cause paths until we reach a self-caused flopflip: that we define as the *ultimate cause*.

Lemma 3.7 *In trees and rings of length not a multiple of 3, there are at most $O(n^2)$ flopflip moves between successive progress moves.*

PROOF SKETCH. If we follow the cause paths, a node can be the ultimate cause of a u-to-v flopflip at most twice, unless there is a cycle which alternates between two matched nodes and one unmatched node (and hence has length a multiple of 3). $\Diamond$

Since the corresponding state changes by matched nodes must be of the same order as the number of flopflips, we get:

Theorem 2 *The 1-maximal matching algorithm converges in $O(n^4)$ moves in trees and rings of length not a multiple of* 3.

4 Performance and Optimality

It is well known that a maximal matching has cardinality at least half that of a maximum matching. A better bound holds for 1-maximal matchings.

Lemma 4.1 *Any 1-maximal matching has at least* 2/3 *the cardinality of a maximum matching.*
PROOF SKETCH. Consider a 1-maximal matching M and and a maximum matching N. Each component C of the graph induced by $M \cup N$ is a path or even cycle; one can easily show that the number of M-edges is at least 2/3 the number of N-edges in C. $\Diamond$

We show next that in a ring of length a multiple of 3, no anonymous 1-maximal matching algorithm is possible. (The maximum matching algorithm for bipartite graphs given in [1] is not anonymous—the claim in their introduction is incorrect.)

Lemma 4.2 *No anonymous self-stabilizing algorithm on a ring of order n a multiple of* 3 *can guarantee more than the smallest maximal matching (which has cardinality $n/3$).*
PROOF SKETCH. Use the standard symmetry argument: the central daemon can ensure that the ring remains symmetric under rotation through three nodes. $\Diamond$

References

[1] S. Chattopadhyay, L. Higham, and K. Seyffarth. Dynamic and self-stabilizing distributed matching. In *Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing*, pages 290–297. ACM Press, 2002.

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

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

[4] S. Dolev. *Self-Stabilization*. MIT Press, 2000.

[5] W. Goddard, S. T. Hedetniemi, D. P. Jacobs, and P. K. Srimani. Fault tolerant algorithms for orderings and colorings. In *Proceedings of the IPDPS Workshop on Advances in Parallel and Distributed Computational Models (APDCM04)*, 2004.

[6] S. M. Hedetniemi, S. T. Hedetniemi, D. P. Jacobs, and P. K. Srimani. Self-stabilizing algorithms for minimal dominating sets and maximal independent sets. *Comput. Math. Appl.*, 46(5-6):805–811, 2003.

[7] S.-C. Hsu and S.-T. Huang. A self-stabilizing algorithm for maximal matching. *Inform. Process. Lett.*, 43:77–81, 1992.

[8] Z. Shi, W. Goddard, and S. T. Hedetniemi. An anonymous self-stabilizing algorithm for 1-maximal independent sets in trees. *Inform. Process. Lett.*, 91:77–83, 2004.

[9] S. Shukla, D. Rosenkrantz, and S. Ravi. Observations on self-stabilizing graph algorithms for anonymous networks. In *Proceedings of the Second Workshop on Self-Stabilizing Systems*, page 7.17.15, 1995.

[10] G. Tel. *Introduction to Distributed Algorithms, Second Edition*. Cambridge University Press, Cambridge UK, 2000.