

Self-Stabilizing Distributed Algorithm for Strong Matching in a System Graph

Wayne Goddard¹ and Stephen T. Hedetniemi¹ and David P. Jacobs¹ and Pradip K. Srimani¹

Department of Computer Science, Clemson University, Clemson, SC 29634-0974

Abstract. We present a new self-stabilizing algorithm for finding a maximal strong matching in an arbitrary distributed network. The algorithm is capable of working with multiple types of demons (schedulers) as is the most recent algorithm in [1, 2]. The concepts behind the algorithm, using Ids in the network, promise to have applications for other graph theoretic primitives.

1 Introduction

The traditional approach to building fault-tolerant, distributed systems uses *fault masking*. It is *pessimistic* in the sense that it assumes a worst case scenario and protects the system against such an eventuality. Validity is guaranteed in the presence of faulty processes, which necessitates restrictions on the number of faults and on the fault model. But fault masking is not free; it requires additional hardware or software, and it considerably increases the cost of the system. This additional cost may not be an economic option, especially when most faults are transient in nature and a temporary unavailability of a system service is acceptable for a short period of time. The paradigm of *Self-stabilization* can cope with the transient faults in a very elegant and cost effective way to design fault tolerant protocols for networked computer systems. Self-stabilization is an *optimistic* way of looking at system fault tolerance, because it provides a built-in safeguard against transient failures that might corrupt the data in a distributed system. Although the concept was introduced by Dijkstra in 1974 [3], and Lamport [4] showed its relevance to fault tolerance in distributed systems in 1983, serious work only began in the late nineteen-eighties. A good survey of self-stabilizing algorithms can be found in [5]. Herman's bibliography [6] also provides a fairly comprehensive listing of most papers in this field. Because of the size and nature of many ad hoc and geographically distributed systems, communication links are unreliable. The system must therefore be able to adjust when faults occur. But 100% fault tolerance is not warranted. The promise of self-stabilization, as opposed to fault masking, is to recover from failure in a reasonable amount of time and without intervention by any external agency. Since the faults are transient (eventual repair is assumed), it is no longer necessary to assume a bound on the number of failures.

A fundamental idea of self-stabilizing algorithms is that the distributed system may be started from an arbitrary global state. After a finite amount of time

the system reaches a correct global state, called a *legitimate* or *stable* state. An algorithm is self-stabilizing if (i) for any initial illegitimate state it reaches a legitimate state after a finite number of node moves, and (ii) for any legitimate state and for any move allowed by that state, the next state is a legitimate state. A self-stabilizing system does not guarantee that the system is able to operate properly when a node continuously injects faults in the system (Byzantine fault) or when communication errors occur so frequently that the new legitimate state cannot be reached. While the system services are unavailable when the self-stabilizing system is in an illegitimate state, the repair of a self-stabilizing system is simple; once the offending equipment is removed or repaired the system provides its service after a reasonable time.

A distributed system can be modeled with an undirected graph $G = (V, E)$, where V is a set of n nodes and E is a set of m edges. If $i \in V$, then $N(i)$, its *open neighborhood*, denotes the set of nodes to which i is adjacent, and $N[i] = N(i) \cup \{i\}$ denotes its *closed neighborhood*. Every node $j \in N(i)$ is called a *neighbor* of node i . Throughout this paper we assume G is connected and $n > 1$. In a self-stabilizing algorithm, a node changes its local state by making a *move* (a change of local state). The algorithm is a set of rules of the form “**if** $p(i)$ **then** M ”, where $p(i)$ is a predicate and M is a move. A node i becomes *privileged* if $p(i)$ is true. When a node becomes privileged, it may execute the corresponding move. We assume a serial model in which no two nodes move simultaneously. A central daemon selects, among all privileged nodes, the next node to move. If two or more nodes are privileged, one cannot predict which node will move next. Multiple protocols exist [7–9] that provide such a scheduler. Our proposed algorithm can easily be combined with any of these protocols to work under different schedulers as well.

There has been a recent spurt of activities in designing self-stabilizing distributed algorithms for different graph theoretic problems in the context of ad hoc networks [10, 11]. In this paper we present the first self-stabilizing algorithm for finding the maximal the strong (induced) matching in an arbitrary graph. Given an undirected graph $G = (V, E)$, a **matching** is defined to be a subset M of edges $(i, j) \in E$, where $i, j \in V$ such that for all nodes $i \in V$ at most one edge of M is incident on i . If \mathcal{M} is a matching, and the edge $(i, j) \in \mathcal{M}$, we say that the nodes i and j are saturated by the edge (i, j) . A matching \mathcal{M} is called **strong** if every saturated node is adjacent to only one saturated node [12]. These strong or induced matchings in graphs have, in addition to theoretical importance, many practical significance, especially when the graph denotes the communication network [13].

2 Self-Stabilizing Maximal Strong Matching Algorithm

Our algorithm requires that every node has a unique ID. We will sometimes use i interchangeably to denote a node and the node’s ID. We assume there is a total ordering on the IDs.

In the algorithm, each node i maintains only a pointer $P(i)$: the value of $P(i)$ is either another node, or one of two special values: **Open** or **Unav**(for “unavailable”).

Definition 1. A node i is called **available** iff $P(i) \neq \text{Unav}$.

We use a total ordering \preceq on the set of undirected edges (or more precisely on the set of pairs of nodes) called the *lexicographic order* (imposed by the total ordering of the unique node IDs). If edges $e = \{u, v\}$ and $f = \{x, y\}$ do not have a node in common, then the one incident with the smallest node is the smaller; if they share a node, then they are ranked by the other end.

Definition 2. If $e = \{u, v\}$ and $f = \{x, y\}$, then $e \prec f$ if and only if either $\min(e) < \min(f)$ or $\min(e) = \min(f)$ and $\max(e) \leq \max(f)$. If $e \preceq f$ and $e \neq f$ then we will write $e \prec f$.

Remark 1. For any node j , when the value of the variable $P(j)$ is an adjacent node (i.e., $P(j) \notin \{\text{Open}, \text{Unav}\}$), the pair $\{j, P(j)\}$ denotes an edge incident on node j .

Definition 3. For any node i , we define a special edge $A(i)$ as

$$A(i) = \min\{ \{j, P(j)\} : j \in N(i) \wedge P(j) \notin \{\text{Open}, \text{Unav}\} \},$$

Remark 2.

- The comparison of edges in Definition 3 is done using the ordering \prec given in Definition 2.
- $A(i)$ is the smallest edge incident to a neighbor of i , as shown by the pointers of its neighbors.
- For any node i in an arbitrary global state, $A(i)$ may not exist. $A(i)$ is unique (when it exists) and is computable by node i using local information.

Definition 4. The *minimum neighbor* of any node i , that is available, is defined as

$$B(i) = \min\{ j : j \in N(i) \wedge P(j) \neq \text{Unav} \}$$

Remark 3. For any node i in an arbitrary global state, $B(i)$ may not exist.

For example, consider the graph shown in Figure 1 (where the values inside the nodes give the nodes’ IDs and the values outside the value of P). For the node 2, $A(2) = \{1, 4\}$ and $B(2) = 4$.

Definition 5. We define the **consistent** value $Q(i)$ for a node i as follows:

$$Q(i) = \begin{cases} \text{Unav} & \text{iff } A(i) \text{ exists and either } B(i) \text{ does not exist or } A(i) \prec \{i, B(i)\} \\ B(i) & \text{iff } B(i) \text{ exists and either } A(i) \text{ does not exist or } \{i, B(i)\} \preceq A(i) \\ \text{Open} & \text{iff neither } A(i) \text{ nor } B(i) \text{ exists} \end{cases}$$

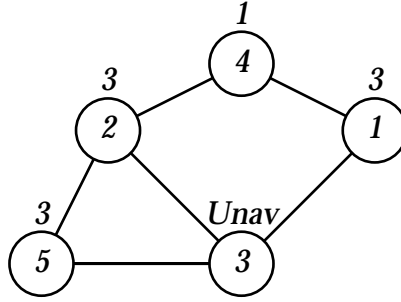


Fig. 1. An example

Note that the value $Q(i)$ can be computed by the node i (i.e., it uses only local information).

Definition 6. A node i in any given state is consistent iff $P(i) = Q(i)$; that is, for a node i to be consistent its $P(i)$ value must be unavailable if there is an edge incident with a neighbor that is smaller than any that it could be in; or must point to the smallest available neighbor if there is one; otherwise it must be open.

The self-stabilizing algorithm SM for maximal strong matching in a given graph consists of one rule as shown in Figure 2. Thus, a node is **privileged** if $P(i) \neq Q(i)$. If it executes, then it sets $P(i) = Q(i)$.

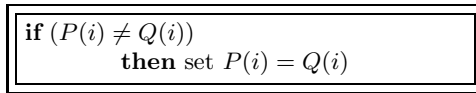


Fig. 2. Algorithm SM: Maximal Strong Matching Algorithm

3 Correctness at Convergence

Assume that the algorithm SM terminates in finite time; that is, the algorithm reaches a global state where all nodes are consistent, i.e., no node is privileged.

Definition 7. A global state is called legitimate iff no node is privileged.

Lemma 1. In a legitimate state, for any node i , if $P(i) = j$ then $P(j) = i$.

Proof. We claim that if $P(j) = \text{Unav}$ or $P(j) < i$, then i is privileged. For, if $P(j) = \text{Unav}$ then $B(i) \neq j$ so that $P(i) \neq Q(i)$; while if $P(j) < i$ then since $A(i) \preceq \{j, P(j)\}$ it follows that $Q(i) = \text{Unav}$ or $Q(i) \leq P(j)$, so that $P(i) \neq Q(i)$. Further, we claim that if $P(j) = \text{Open}$ or $P(j) > i$, then j is privileged. For, since $B(j) \leq i$, it follows that $Q(j) = \text{Unav}$ or $Q(j) \leq i$.

Remark 4. In a legitimate state, the set of edges $\{\{i, j\}; i, j \in V \wedge P(i) = j \wedge P(j) = i\}$ defines a matching \mathfrak{M} in the graph.

Lemma 2. *The set of edges in the matching \mathfrak{M} induces a collection of isolated edges, i.e., the matching \mathfrak{M} is strong.*

Proof. Suppose adjacent nodes i and j are both in the matching, but the edge between them is not part of the matching. That is, edges $e = \{i, P(i)\}$ and $f = \{j, P(j)\}$ are disjoint. Without loss of generality, $e \prec f$. Then j is privileged, a contradiction.

Lemma 3. *The strong matching \mathfrak{M} is maximal.*

Proof. Suppose an edge $e = (i, j)$ can be added to \mathfrak{M} and it still be a strong matching. Then neither i nor j has a neighbor in \mathfrak{M} . So neither $A(i)$ nor $A(j)$ exists. This means that since i and j are consistent, neither is unavailable; but then $B(i)$ and $B(j)$ exist, and hence both nodes are inconsistent, a contradiction.

4 Convergence or Termination in Finite Time

In this section we show that the algorithm SM terminates in a finite number of moves when it starts from an arbitrary initial state. The algorithm terminates when no node is privileged.

Definition 8. *Let X be a totally ordered set. Consider a sequence \mathfrak{S} of subsets $S(t)$ of X , $t = 1, 2, \dots$, such that each subset is obtained from the previous one by either the addition or deletion of one element. We say such a sequence is **downward** if for all elements i , if i is added at time t and then deleted at time t' , then some element smaller than i is added between time t and t' . That is, if $i \in S(\tau)$ for $t \leq \tau < t'$ but $i \notin S(t-1), S(t')$, then there exists $j < i$ and τ with $t < \tau < t' - 1$ such that $S(\tau) = S(\tau - 1) \cup \{j\}$.*

Lemma 4. *If X is finite, then a downward sequence \mathfrak{S} on X is finite.*

Proof. We prove by induction on $|X|$. Consider the minimum element of X ; call it 1. It might be in or out of $S(1)$. But once added, it cannot be deleted. So define \mathfrak{S}' as the subsequence up to 1's addition, if it exists, and \mathfrak{S}'' as the subsequence after 1's addition. (If 1 is never added then set $\mathfrak{S}' = \emptyset$ and $\mathfrak{S}'' = \mathfrak{S}$.) Then ignore the transition where 1 is deleted, if this occurs, and restrict both subsequences to the set $X - \{1\}$. The result is two downward sequences on the set $X - \{1\}$. Hence, if $M(m)$ denotes the maximum length of a downward sequence for a set of cardinality m , it follows that $M(m) \leq 2 + 2M(m - 1)$.

Definition 9. For any node k , in a given global state, we define the set of edges

$$S_k = \{ \{j, P(j)\} : j \geq k \wedge P(j) < k \}.$$

Remark 5. Note that S_k exists for a node k only when $P(k) \notin \{\text{Open}, \text{Unav}\}$. Also, $0 \leq |S_k| \leq n - 1$.

Definition 10. We say that a node is **static** if it does not execute (irrespective of it is privileged or not). We say a node is **stable** if does not execute at all until termination of the algorithm.

Remark 6. During the interval from the time the algorithm starts from an arbitrary state to the time it converges, a node may be static over a period of time but may not be stable during that period.

Lemma 5. Assume the nodes less than node k , for a given k , are static over a period of time. Then S_k can change only a finite number of times during that period.

Proof. Suppose at some stage $P(i)$ becomes j , with $i \geq k > j$. Since the nodes less than k are stable, it follows that $B(i) = j$ throughout. So for i to change again, it must happen that $A(i)$ changes to a value smaller than $\{i, j\}$.

That is, if at some stage $\{i, j\}$ is added to S_k , then before it is deleted some smaller edge must be added to S_k . (Note that if at the start $P(i) = j'$ with $j' < k$, then we assume the change in S_k occurs in two steps: add $\{i, j\}$ and then delete $\{i, j'\}$.) Thus, the sequence of S_k is a downward sequence. By Lemma 4, S_k can change only a finite number of times.

Lemma 6. Assume the nodes less than k , for a given k , are static over a period of time.

1. If node k is at some stage consistent and available, and later declares itself not available, then in between there was an addition to S_k .
2. If k is available throughout, and v is some neighbor of k that is at some stage consistent and available, and later declares itself not available, then in between there was an addition to S_k .

Proof. 1. If node k declares itself not available, a better edge must have been created in its neighborhood since node k was consistent. Hence there was an addition to S_k .

2. If k is available, then at the moment of declaring itself not available the node v must see an edge smaller than $\{v, k\}$, which did not exist when it was consistent. Hence there must have been an addition to S_k .

Lemma 7. Assume the nodes less than k are stable. Then k moves a finite number of times.

Proof. There cannot be an infinite cycle of node k doing just the following types of moves: **Unav** to **Open**; **Open** to a value and/or decrease. So consider each time that k declares itself unavailable, increases value or changes from a value to being open. The latter two can only occur if some neighbor declares itself not available. Hence each such move involves either k or one of its neighbors declaring itself unavailable. By Lemmas 5 and 6, this can occur only a finite number of times. Hence k can be privileged only a finite number of times.

Theorem 1. *Starting from an arbitrary initial state, the algorithm terminates in finite amount of time.*

Proof. We use induction and Lemma 7; it follows that nodes $\{1, \dots, k\}$ can be privileged (or make a move) only a finite number of times. Hence the algorithm terminates.

5 Complexity Analysis

We note that in our application, the downward sequences have the additional property that if i is deleted because of j 's addition, then i cannot be re-added until j is deleted. It can readily be shown that such sequences have length $O(|X|^2)$. However, this provides no improvement in the bound on the running time of the overall algorithm—which is $O(n^n)$ since no state can repeat.

The algorithm we present does indeed have exponential running time. Consider the following example. Take any graph G and add three new nodes x_3, x_2, x_1 such that x_3 is adjacent to all nodes, while x_1 and x_2 are adjacent only to each other and to x_3 , and such that these three nodes have the smallest IDs and $x_3 > x_2 > x_1$. Call the resulting graph G' .

Assume all nodes start as **Unav**. Then assume the demon proceeds as follows:

- (1) fire G until it stabilises;
- (2) fire x_1 (so goes to **Open**) and then x_3 (so points to x_1);
- (3) fire all nodes in G (so go to **Unav**);
- (4) fire x_2 (so points to x_1) and then x_3 (so goes to **Unav**);
- (5) fire G until it stabilises.

If $M(G)$ denotes the maximum number of steps on graph G assuming all nodes start as unavailable, then it follows that $M(G') \geq 2M(G) + 4$. By repeating this construction, it follows that running time can be at least $2^{n/3}$.

6 Acknowledgment

This work was supported by NSF grant # ANI-0218495 and Srimani's work was partly supported also by NSF grant # ANI-0073409.

References

1. M Gradinariu and S Tixeuil. Self-stabilizing vertex coloration of arbitrary graphs. In *4th International Conference On Principles Of Distributed Systems, OPODIS'2000*, pages 55–70. Studia Informatica Universalis, 2000.

2. S Dolev and JL Welch. Crash resilient communication in dynamic networks. *IEEE Transactions on Computers*, 46:14–26, 1997.
3. E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.
4. L. Lamport. Solved problems, unsolved problems, and non-problems in concurrency. In *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing*, pages 1–11, 1984.
5. M. Schneider. Self-stabilization. *ACM Computing Surveys*, 25(1):45–67, March 1993.
6. T. Herman. A comprehensive bibliography on self-stabilization, a working paper. *Chicago J. Theoretical Comput. Sci.*, <http://www.cs.uiowa.edu/ftp/selfstab/bibliography>.
7. G Antonoiu and PK Srimani. Mutual exclusion between neighboring nodes in an arbitrary system graph tree that stabilizes using read/write atomicity. In *Euro-Par'99 Parallel Processing, Proceedings LNCS:1685*, pages 823–830, 1999.
8. J Beauquier, AK Datta, M Gradinariu, and F Magniette. Self-stabilizing local mutual exclusion and daemon refinement. In *DISC00 Distributed Computing 14th International Symposium, Springer LNCS:1914*, pages 223–237, 2000.
9. M Nesterenko and A Arora. Stabilization-preserving atomicity refinement. In *DISC99 Distributed Computing 13th International Symposium, Springer LNCS:1693*, pages 254–268, 1999.
10. J. R. S. Blair and F. Manne. Efficient self-stabilizing algorithms for tree networks. In *Proceedings of ICDCS-2003*, Rhode Island, 2003.
11. S. T. Hedetniemi, D. P. Jacobs, and P. K. Srimani. Fault tolerant distributed coloring algorithms that stabilize in linear time. In *Proceedings of the IPDPS-2002 Workshop on Advances in Parallel and Distributed Computational Models*, pages 1–5, 2002.
12. K. Cameron. Induced matchings. *Discrete Applied Mathematics*, 24:97–102, 1989.
13. M. C. Golumbic and M. Lewenstein. New results in induced matchings. *Discrete Applied Mathematics*, 101(1-3):157–165, 2000.