1

Abstract

Finding a maximal or maximum matching in a graph is a well-understood problem for which efficient sequential algorithms exist. Applications of matchings in distributed settings make it desirable to find self-stabilizing asynchronous distributed solutions to these problems. We first present a self-stabilizing algorithm for finding a maximal matching in a general anonymous network under read/write atomicity with linear round complexity. This is followed by a self-stabilizing algorithm for finding a maximum matching in a bipartite network under composite atomicity with quadratic round complexity. These results represent significant progress in the area of distributed algorithms for matchings.

Dynamic and Self-Stabilizing Distributed Matching

Subhendu Chattopadhyay, Lisa Higham, Karen Seyffarth
University of Calgary

1 Introduction

A *matching* in a graph $G(V, E)$ is a subset M of E in which no pair of edges is adjacent. The matching M is *maximal* if no proper superset of M is a matching, and is *maximum* if there in no matching in G with size strictly larger than the size of M.

Several distributed applications employ matchings as a basic component of the system. In many cases it would be useful to construct the matching dynamically as the system configuration changes. For example, in a system of processors consisting of a collection of clients each connected to some subset of servers, a maximum matching would pair as many clients with servers as possible. The system would be even more useful if the pairing happens dynamically, so that when a client is finished, it is removed from the system and another client can be matched with the free server. If the servers are stateless, we may even require that the system be capable of reconfiguring on the fly so that some client without access to the free server, but with access to a busy server, could nevertheless gain service by dynamically adjusting the current pairing. Ideally, any large system of this type would also be self-stabilizing so that it would not require initialization and could automatically recover from transient faults.

1.1 Distributed Systems and Self-Stabilization

A distributed system is a collection of processors that communicate with each other to solve a common problem. Often the purpose of a distributed system is to solve problems that are inherently distributed, for example resource allocation, consensus. In a distributed system the processors are modelled as state machines. A distributed algorithm determines the state transition function of the processors. A state transition by a processor is also termed as a *move* or a *step*. The collection of the states of the processors determines the system configuration. The final output of a distributed system is manifested by the states of the individual processors.

A distributed system is more useful if it can withstand transient fault. A transient fault may corrupt the states of the processors rendering the system in an arbitrary configuration. One approach in building distributed systems capable of withstanding transient fault is self-stabilization. A self-stabilizing algorithm is a non-terminating algorithm which guarantees

*This research was supported in part by the Natural Sciences and Engineering Research Council of Canada

that no matter what configuration the system is started, after a finite amount of computation, the system will reach a legitimate configuration. Once it reaches a legitimate configuration, it always stays in some legitimate configuration in a fault free execution.

Formally, a system is *self-stabilizing* [4] if it satisfies:

1. (Closure): Once the system is in some configuration in a predefined set of legitimate configurations, all subsequent configurations are legitimate, and

2. (Convergence): When started in any configuration, after a finite amount of computation it reaches some legitimate configuration.

A transient fault may move a system into an arbitrary configuration. If the system is self-stabilizing, it will recover without requiring any intervention. Self-stabilizing algorithms were first introduced by Dijkstra [3] in his well-known self-stabilizing mutual exclusion algorithm. Dolev presents much of the previous work on self-stabilization is his book [4] on the subject.

Throughout this paper, a network of processors is modelled as a graph $G(V, E)$. Each node corresponds to a processor, and edges denote neighbouring processors. A processor can read the variables (or state) of each of its neighbours, but can only write to its own variables. We refer to this as the *locally shared memory model*.

The existence or complexity of distributed algorithms for a particular problem depends on several additional characteristics of the model. The computation of the algorithm proceeds under the control of a scheduler. A *distributed scheduler* chooses, for each step, some non-empty subset of processors. In that step each processor simultaneously executes its next move. Under a *centralized scheduler* this subset of processors always has size one. The *atomicity* determines the granularity of an indivisible move of a processor. Under *composite atomicity*, in one indivisible move, a processor reads the state of each of its neighbours, performs local computation and changes its own state. Under *read/write atomicity*, a move by a processor consists of only one read or one write or some internal action. So under read/write atomicity, a processor has to read the states of its neighbours into its own local variable and take decisions based on the values of these local variables which may be different than the actual states of the neighbours. While read/write atomic algorithms are more realistic, they are more difficult to design and analyze. An algorithm may rely on distinct identifiers for each processor, or it may work even for an *anonymous network* where processors are indistinguishable.

1.2 Previous Matching Algorithms

Finding maximal and maximum matchings are well studied problems, for which several efficient centralized algorithms [6, 9] exist for general and constrained families of graphs. Much less work has been done to solve either of these problems in a distributed environment. The few distributed algorithms for matching that we are aware of are self-stabilizing.

Hsu and Huang [10] present a self-stabilizing distributed algorithm for finding a maximal matching in a general network. Their algorithm assumes composite atomicity and a centralized scheduler. It does not require any knowledge of the network size. It works for anonymous networks, although it implicitly assumes some kind of labelling that allows a processor to determine where its neighbours are pointing. The algorithm has an upper bound of $O(n^3)$ state changes, where n is the number of nodes in the system. Subsequent analysis of the same algorithm improves this upper bound, first to $O(n^2)$ (Tel [14]) and then to $\Theta(2m+n)$ (Hedetniemi, Jacobs and Srimani [8]) where m is the number of edges and n is the number of nodes in the system. Gradinariu and Johnen [7] present a randomized algorithm to assign distinct identifiers within distance two in an anonymous network under a distributed scheduler with composite atomicity. They then use this to extend the Hsu-Huang maximal matching algorithm to work under a distributed scheduler.

Karaata and Saleh [11] present a self-stabilizing algorithm for finding a maximum matching in a tree. This algorithm also assumes composite atomicity and a centralized scheduler, but identifiers are not needed. The algorithm uses $O(n^4)$ state changes in the worst case.

1.3 Contributions of this Paper

The composite atomicity used in previous matching algorithms contrasts with the weaker and more realistic read/write atomicity. Our goal is to devise efficient self-stabilizing algorithms for both maximal and maximum matching using only read/write atomicity in anonymous networks. This paper presents two main algorithms that make substantial progress toward this goal and improve upon previous work in various ways. Since a simple symmetry breaking argument shows that a deterministic matching (maximal or maximum) algorithm is impossible for anonymous networks under read/write atomicity, our approach employs randomization.

Our first contribution is a maximal matching algorithm for arbitrary anonymous networks under read/write atomicity. Our matching algorithm requires each processor to have a locally distinct label that distinguishes it from its neighbours within distance two, so we also present a randomized self-stabilizing read/write atomic algorithm that efficiently assigns these locally distinct labels.

Our second contribution is a maximum matching algorithm on anonymous bipartite graphs assuming that each processor knows its bipartition and an upper bound on network size. This algorithm takes a quadratic number of rounds to stabilize. Here we fall short of our goal because our solution only works for bipartite graphs, it uses composite atomicity, and the bipartition must be known in advance. To overcome the last problem we can use fair composition with a colouring technique that computes the bipartition. Sur and Srimani [13] present a self-stabilizing algorithm for coloring a bipartite graph. This algorithm needs a distinguished processor (a leader) and works under central scheduler. Our current colouring algorithm works under read/write atomicity and distributed scheduler but relies on distinct identifiers, which, in turn, can be created with randomization [4, pp. 110-118].

1.4 Terminology, Tools, and Outline for this Paper

The time complexity of a self-stabilizing algorithm is some measure of the time until the system converges to a legitimate configuration. In this paper we measure the round complexity. A *round of computation* is the shortest partial computation in which each enabled processor executes at least one full iteration of its algorithm.[1]

Fair composition is a technique introduced by S. Dolev *et al.*[5] that combines self-stabilizing algorithms for subproblems into a self-stabilizing solution for a more complex problem. Let A and B be two self-stabilizing algorithms for a distributed system. Algorithm A has a restricted closure property, so that once the system reaches a final configuration processors do not change state. Algorithm A has a set of variables that can be read by algorithm B. Algorithm B has a set of variables that are used only by algorithm B. Let C_A and F_A be the set of all configurations and the set of final configurations respectively, as determined by the variables of algorithm A only. Let C_B and F_B be the set of all configurations and the set of final configurations respectively, as determined by the variables of algorithms A and B. Let $C_B \subseteq F_A$. If the processors of the distributed system run the algorithms A and B alternately, after a finite amount of computation algorithm A will reach some configuration in F_A and will remain in that configuration. Behavior of algorithm B is unpredictable up to this point. But algorithm B will converge to some state in F_B after a finite amount of computation once algorithm A converges to some configuration in F_A. Hence starting from any initial configuration in C_A, the system will converge to some final configuration in F_B within finite amount of computation.

Our main algorithms are constructed by the fair-composition technique, which we assume without proof.

In this paper $N(x)$ denotes the set of processors adjacent to processor x, and $N[x] = N(x) \cup \{x\}$. The number of processors in the network is denoted by n and the upper bound on n is denoted by N. The *distance* between two processors x and y, denoted $d(x, y)$, is the number of edges in a shortest path between x and y. For a processor x,

$$N_2(x) = \{y \mid 1 \leq d(x, y) \leq 2\}$$

A network has *locally distinct labels within distance two* if any two different processors x and y, that are within distance two, have different labels.

Section 2 presents a maximal matching algorithm. Sections 3 contains a randomized algorithm for assigning locally distinct labels. Sections 4 contains a maximum matching algorithm for bipartite graphs. Section 5 contains a 2-colouring algorithm for bipartite graphs. Comments and future work are presented in Section 6.

2 Maximal Matching

Consider any deterministic algorithm for an anonymous network with read/write atomicity, with all processors in the same initial states. There is a scheduler (for example round-robin) such that after each round the processors are again in identical states. Therefore it is impossible to break symmetry in this situation. Since any configuration that is a maximal

[1] In most distributed computing literature, a round of computation is defined as the shortest partial computation in which each enabled processor executes at least one atomic step. For the composite atomicity algorithms in this paper, both definitions agree. However, for the read/write atomic algorithms in this paper, a processor performs as many as $\Delta + 1$ atomic operations in one complete iteration, where Δ is the maximum degree of the network.

matching requires at least two distinct edge states, and edge states correspond to pairs of node states, symmetry breaking is required. Thus it is impossible to solve the maximal matching problem in this model. This impossibility can be circumvented if the processors have locally distinct labels.

Model: The model of computation is locally shared memory under read/write atomicity with a distributed scheduler. Processors do not need any information about the network topology. The network has locally distinct labels within distance two. Labels are assigned from the set of positive integers and are stored in the shared variables lid.

Algorithm for processor x: Processor x has a shared variable called pref that takes value from the set of non-negative intergers. It also has a local variable S_x, which is a set of lid's.

`do forever`
    $S_x \leftarrow \emptyset$
    `for all` $y \in N(x)$
        `if` $y.\text{pref} = 0$ or $y.\text{pref} \geq x.\text{lid}$
            $S_x \leftarrow S_x \cup \{y.\text{lid}\}$
    `if` $S_x = \emptyset$
        $x.\text{pref} \leftarrow 0$
    `else`
        $x.\text{pref} \leftarrow min\{S_x\}$

Proof of correctness: Let $M = \{xy \mid x.\text{pref} = y.\text{lid} \text{ and } y.\text{pref} = x.\text{lid}\}$.

Lemma 2.1. *The set M is a matching.*

Proof. A processor can have at most one non-zero preference at a time. Hence there can never be two adjacent edges in M, and so M is a matching. □

Theorem 2.1. *Starting from an arbitrary configuration, the system stabilizes to a maximal matching within $\frac{n}{2}$ rounds.*

Proof. After one round, $x.\text{pref} = 0$ or $x.\text{pref} \in N(x)$ for each processor x. A processor x is a *local minimum* if $x.\text{lid} < y.\text{lid}$ for all $y \in N_2(x)$. Suppose x is a local minimum, and let y be the processor with minimum label in $N(x)$. Since x is a local minimum, $x.\text{lid} < z.\text{lid}$ for all $z \in N[y] - \{x\}$. Because $y.\text{lid} < u.\text{lid}$ for all $u \in N(x) - \{y\}$, $x.\text{pref} = y.\text{lid}$ after the second round and this never changes. Furthermore, $y.\text{pref} = x.\text{lid}$ after the second round (since x is a local minimum) and this never changes. After the second round, let $z \in N(x) - \{y\}$. Then $z.\text{lid} > y.\text{lid} = x.\text{pref}$, so $x.\text{lid} \notin S_z$, and $z.\text{pref}$ will never be set to $x.\text{lid}$. Similarly if $u \in N(y) - \{x\}$, then $u.\text{lid} > x.\text{lid} = y.\text{pref}$, so $y.lid \notin S_u$, and $u.\text{pref}$ will never be set to $y.\text{lid}$.

Therefore, after the first two rounds, each local minimum is matched with its neighbour of minimum id and these matches never change. Also no unmatched neighbour of a matched processor will try to prefer a matched processor. The algorithm now proceeds as if the matched processors have been deleted from the system. It follows by induction that a maximal matching will emerge in at most $\lfloor n/2 \rfloor - 1$ more rounds. $\square$

Observation: If the nodes of a graph has locally distinct labels within distance two and the labels are chosen from a totally ordered set, the graph will have local minimums. The algorithm presented above finds the particular maximal matching given by the following graph theoretic scheme.

Match a local minimum with its neighbour holding the smallest label. Remove the matched nodes and incident edges. The resulting graph has the same property as the original graph. Hence this process can be repeated until there are no more edges. At this point the set of matched edges is a maximal matching.

Maximal matching is a local property in a graph. So processors need only local knowledge to solve this problem. Ideally a self-stabilizing algorithm should need only constant number of rounds to converge under distributed scheduler which schedules all enabled processors in every round. But the current algorithm finds the particular maximal matching defined by the graph theoretic scheme presented above. To find this particular maximal matching, processors may need to propagate information throughout the entire network and hence may need a linear number of rounds to converge. An example where the algorithm needs $n/2$ rounds to converge is given below.

Let the network be a path of processors labelled $1, \ldots, n$, where processor labelled i is adjacent to processor labelled $i+1$ for $1 \leq i \leq n-1$. Let all the processor start with preference value zero. Then the following schedule will take $n/2$ rounds to stabilize to maximal matching.

Step one: $n, n-1, \ldots, 4, 3, 1$

Step two: $n, n-1, \ldots, 5, 4, 2$

This is end of the first round and processors labelled 1 and 2 are matched with each other and are no longer enabled. The system can now be assumed to be comprised of only processors labelled $n, n-1, \ldots, 4, 3$.

Step three: $n, n-1, \ldots, 5, 3$

At this point we are exactly in the same state in the reduced system as the original system after step one. This can be repeated $n/2$ times to give an $n/2$ round execution.

Once a self-stabilizing algorithm converges to maximal matching, a transient fault may leave the system in a configuration which is not a maximal matching. If the fault is local, ideally it should be contained and fixed locally. That means the disturbance created by the local fault should not propagate more than a constant distance from the region of the fault. But present algorithm finds a particular maximal matching. So the disturbance may propagate to the entire network and re-convergence can be as costly as $n/2$ rounds.

3 Randomly Generating Locally Distinct Labels

The maximal matching algorithm presented in Section 2 requires locally distinct labels within distance two. It follows directly from the previous symmetry breaking observation that this label distribution cannot be done deterministically with read/write atomicity. A randomized self-stabilizing algorithm that assigns locally distinct labels within distance two is presented below. A fair composition of these two algorithms solves the maximal matching problem in a general anonymous network with read/write atomicity.

Model: The model of computation is an anonymous network with locally shared memory and a distributed scheduler under read/write atomicity.

Data structures: Each processor x maintains a shared label called lid and a shared variable δ, which are positive integers. Processors x has a shared array H_x of sets of labels, where each set is indexed by a processor in $N(x)$. Let $N(x) = \{y_1, \ldots, y_m\}$. Then

$$H_x[y_i] = \{x.\text{lid},\ y_1.\text{lid}, \ldots, y_{i-1}.\text{lid},\ y_{i+1}.\text{lid}, \ldots, y_m.\text{lid}\}$$

When processor y_i reads H_x it only reads $H_x[y_i]$.

Processor x has local variables Δ holding a positive integer, A_x holding a set of labels, and P_x holding a multiset of labels. Let $\text{mult}(S, x)$ denote the multiplicity of x in multiset S. Multisets $S + x$ and $S - x$ are similar to S except for the multiplicity of x. The multiplicity of x in $S + x$ is $\text{mult}(S, x) + 1$. The multiplicity of x in $S - x$ is $\max\{0, \text{mult}(S, x) - 1\}$.

When multiset S is assigned to a set T, the operation reduces the multiplicity of all elements of S to one and the operation is denoted by $T \leftarrow S$.

Algorithm for processor x

`do forever`
- $\delta \leftarrow |N(x)|$
- $A_x \leftarrow \emptyset$
- $P_x \leftarrow \emptyset$
- $\Delta \leftarrow \delta$
- `for all` $y \in N(x)$
  - $A_x \leftarrow A_x \cup H_y[x]$
  - $P_x \leftarrow P_x + y.\text{lid}$
  - $\Delta \leftarrow max\{\Delta, y.\delta\}$
- `if` (`there exists` $z \in A_x$ `such that` $z = x.\text{lid}$) `then`
  - randomly choose $x.\text{lid}$ uniformly from 1 to $9\Delta^2(\Delta + 1)$ that is not in A_x
- $P_x \leftarrow P_x + x.\text{lid}$
- `for all` $y \in N(x)$
  - $H_x[y] \leftarrow P_x - y.\text{lid}$

Analysis and correctness

Lemma 3.1. *After two rounds, the a processor's local variable Δ holds the maximum degree among all processors in its closed neighbourhood.*

Proof. After the first round, the shared variable δ of a processor always holds the size of its neighbourhood. In the second round, every processor will correctly compute the maximum degree in its closed neighbourhood and store it in Δ and this value will be correct from then on. $\square$

Theorem 3.1. *Starting from an arbitrary configuration, the system will stabilize to have locally distinct labels within distance two within an expected* $2\log_3 n + 6$ *rounds.*

Proof. By Lemma 3.1, a processor's Δ value is correct after two rounds. The rest of the proof assumes that the Δ value is correct.

For processor x, define

$$L(x) = \bigcup_{y \in N_2(x)} \{y.lid\} \bigcup_{y \in N(x)} H_y[x] \bigcup A_x$$

$L(x)$ is the set consisting of: (i) the labels of all neighbours of x within distance two; (ii) the labels in the row for x in the arrays of each of x's neighbours; (iii) the labels in x's local set A_x. The size of $L(x)$ is less than $3\Delta^2$. Define x to be *secure* if $x.lid \notin L(x)$; otherwise x is *insecure*.

Observe that

1. Processor x does not choose a new label as long as it is secure. If all processors are secure then the system has stabilized to locally distinct labels within distance two, and labels will no longer change.

   Follows directly from the condition for the choice operation and the definition of secure.

2. If an insecure processor chooses and is still insecure, it causes at most Δ secure processors to become insecure. This happens with probability less than $\frac{3\Delta^2}{9\Delta^2(\Delta+1)}$.

   Let $N(x) = \{y_1, \ldots, y_m\}$ for some processor x. Then $N_2(x) \subset \bigcup_{i=1}^{m} N[y_i]$. If $z_i \in N[y_i]$ is a secure processor, then there can not be another processor $u \in N[y_i]$ with $u.\text{lid} = z_i.\text{lid}$. Hence there can be at most m secure processors $z_1 \in N[y_1], \ldots, z_m \in N[y_m]$ in $N_2(x)$ such that $z_i.\text{lid} = z_j.\text{lid}$ for $1 \leq i, j \leq m$. If processor x is insecure and choses the label $z_1.\text{lid}$, then it remains insecure and makes processors $\{z_1, \ldots, z_m\}$ insecure. Since a processor can have at most Δ neighbours, an insecure processor can make at most Δ secure processors insecure by one choice.

3. An insecure processor cannot remain insecure continuously for more than two rounds without choosing a new label.

   If processor x is insecure because $x.\text{lid} \in A_x$, then processor x either updates A_x and becomes secure or chooses a new label in the current round. If it is insecure because $x.\text{lid} \in H_y[x]$ for some processor $y \in N(x)$, then either y updates $H_y[x]$ and x becomes secure or $x.\text{lid}$ appears in A_x in the next round and x chooses a new label in the same round. If it is insecure because $x.\text{lid} = y.\text{lid}$ for some processor $y \in N_2(x)$, then either x becomes secure because y chooses a new label different from $x.\text{lid}$, or $x.\text{lid}$ appears in $H_z[x]$ for some $z \in N(x)$ and subsequently either x becomes secure or chooses a new label within two rounds.

Define a super-round as two consecutive rounds of computation. Define random variable X_i as the number of insecure processors in the system at the start of super-round $i+1$. If at the begining of super-round $i+1$ there are l insecure processors in the system, by observation 3 above, each one of these processor either becomes secure or chooses a new label within one super-round. When an insecure processor chooses, it can stay insecure with probability less than $\frac{3\Delta^2}{9\Delta^2(\Delta+1)}$. Hence the expected number of processors among those l insecure processors that can stay insecure is less than $l \cdot \frac{3\Delta^2}{9\Delta^2(\Delta+1)}$. Each of these processors can make at most Δ secure processors insecure (Observation 2). Hence

$$\begin{aligned}
E(X_{i+1}|X_i = l) &< l \cdot \frac{3\Delta^2}{9\Delta^2(\Delta+1)} + l \cdot \Delta \cdot \frac{3\Delta^2}{9\Delta^2(\Delta+1)} \\
&= l/3
\end{aligned}$$

$$\begin{aligned}
E(X_{i+1}) &= \sum_{l \geq 0} E(X_{i+1}|X_i = l) \cdot Pr[X_i = l] \\
&< \sum_{l \geq 0} \frac{l}{3} \cdot Pr[X_i = l] \\
&= 1/3 \cdot E(X_i)
\end{aligned}$$

Thus if the system starts with m ($\leq n$) insecure processors, after $\log_3 m$ super-rounds, the expected number of insecure processors will become one. Let $E(i, j), 0 \leq j \leq i \leq n$ denotes the expected number of rounds for the number of insecure processors to go down from i to j. Then $E(m, 1) = \log_3 m$.

$$\begin{aligned}
E(1,0) &= 1 \cdot \left(1 - \frac{3\Delta^2}{9\Delta^2(\Delta+1)}\right) + \frac{3\Delta^2}{9\Delta^2(\Delta+1)} \cdot E(\Delta, 0) \\
E(1,0) &= 1 \cdot \left(1 - \frac{1}{3\Delta+3}\right) + \frac{1}{3\Delta+3} \cdot (E(\Delta,1) + E(1,0)) \\
E(1,0) &= 1 \cdot \left(1 - \frac{1}{3\Delta+3}\right) + \frac{1}{3\Delta+3} \cdot (\log_3 \Delta + E(1,0)) \\
(3\Delta+2) \cdot E(1,0) &= 3\Delta + 2 + \log_3 \Delta \\
E(1,0) &= 1 + \frac{\log_3 \Delta}{3\Delta+2} \\
&< 2
\end{aligned}$$

Thus, if there is only one insecure processor, then within expected two super-rounds all processors will become secure. By observation 1 above, after this time the system has locally distinct labels within distance two and there will be no more choosing operations. Hence the system stabilizes to locally distinct labels within expected $2\log_3 n + 6$ rounds. □

Observation: A fair composition of the label generation algorithm and the maximal matching algorithm solves the maximal matching problem in a general anonymous network under read/write atomicity. In our maximal matching algorithm, we observed that the algorithm is not local if the labels are assigned in an unfavourable manner. This can not happen If the labels are assigned using the label generation algorithm. Intuition behind this argument is that if the disturbance has to propagate a long distance, there has to be a long chain of processors with increasing labels without intervening local minimums. Since the labels are choses randomly from a small set, this can not happen.

4 Maximum Matching

This section presents a self-stabilizing algorithm for finding a maximum matching in a bipartite graph, where each processor knows its bipartition. A self-stabilizing algorithm for 2-colouring a bipartite graph is presented in Section 5. Thus a fair composition of these two algorithms solves the bipartite maximum matching problem.

Model: The model of computation is an anonymous network with locally shared memory, under composite atomicity and a centralized scheduler. The network is a bipartite graph with bipartitions U and V. Each processor knows its bipartition and some upper bound N on the number of processors in the network. Each processor x can determine for each of its neighbours y if y's pointer is null, points to x, or points elsewhere.

4.1 Informal Description of the Algorithm

The protocol is motivated by the well known sequential maximum matching algorithm for bipartite graphs [12, pp. 218-224], from which the following definitions and theorem are taken. Graph theoretic background for this algorithm can be found in most introductory graph theory texts (for example [15]). An *alternating* path in a graph G, with respect to matching M, is a path whose edges are alternately in M and in $(E - M)$. An alternating path is an *augmenting path* if the first and the last nodes are not incident to any edge in M.

Theorem 4.1. *(Berge [2]) A matching M in a graph G is maximum if and only if there is no augmenting path in G with respect to M.*

In the sequential algorithm for finding a maximum matching, some edges (initially none) are matched. If the matching is not maximum, an augmenting path is detected, and augmentation along this path increases the size of the matching by one. The algorithm terminates when no more augmenting paths exist, implying that the matching is as large as possible. To implement this idea in a distributed asynchronous network, the processors themselves must identify non-intersecting augmenting paths and then augment along them.

The state of a processor, x, consists of four variables: preference, choice, augment-status and distance. Each variable can assume a finite number of values implying the state space of a processor is bounded and a processor can be modelled as a finite state machine. For processor x, each of the preference and the choice fields is either null, or is a pointer to one of x's neighbours. Loosely, the preference field is intended to record with which neighbour a processor is currently matched, if any, and the choice field records a possible new match arising from an augmenting path. The augment-status is either silent, requesting, acknowledging or changing, and conveys how a processor is currently participating in trying to realize an augmenting path. The distance field is a non-negative integer less than N, intended to record the distance to an endpoint of an augmenting path. It is used to prevent processors from forming an alternating cycle instead of an alternating path when searching for an augmenting path and to recover from an unsuccessful augmentation process.

In each composite step, a processor reads all its neighbours states and then performs a sequence of four internal steps. Initially, the state of a processor may be arbitrary. However, not all combinations of preference, choice, augment-status and distance could arise during the course of the algorithm, so in Step 1 a processor checks that its state is valid and resets otherwise. Furthermore, some valid states are not consistent with neighbours in other valid states, so in Step 2 a processor resets if it is inconsistent with its neighbours. Step 3 ensures that preferences are mutual and thus form a matching, unless they are in the process of changing due to an augmentation. These three steps ensure that after a few initial rounds of computation, the global state of the network has enough consistency that Step 4 will eventually augment the matching if an augmenting path exists.

For the description of Step 4 it is helpful to imagine that a processor "sends" information to its neighbours and "propagates" information through the network, although this is achieved by a processor's state changes as observed by its neighbours. The protocol is initiated by any unmatched U processor. Such a processor (a source) sends a request to each of its neighbours (necessarily V processors). Each such request propagates along alternating paths that may branch at U processors in the network and forms a tree structure (a requesting tree) rooted at the source. If a requesting branch encounters an unmatched V processor (a sink), then the path through this requesting tree from the sink back to the source is an augmenting path. A sink tries to reserve an augmenting path. To do so, a sink initiates an acknowledgment that is sent toward the source along the alternating path in the requesting tree, reserving the processors on the path as it goes. When an acknowledgment arrives at a branching node of the requesting tree, it prunes away all the node's siblings as the acknowledgment is forwarded toward the source. Thus, eventually, only one sink succeeds in marking a path all the way to the source of its requesting tree, and the tree has been pruned to a single augmenting path. When an acknowledgment reaches a source, the source initiates a change demand that travels back along this path to the sink, updating the matching to the augmented matching as it goes.

While a tree is growing through request propagation and being pruned by returning acknowledgments, its nodes only know local information about the tree structure. Specifically, each processor knows its parent, and as a potential augmenting path is being constructed, each U processor knows its chosen child. (Parents and children are defined by the state of a processor and the states of its neighbours.) Nodes in the pruned subtrees of a tree have no immediate way of detecting that they are no longer attached to the source. Parts of these subtrees may continue to grow because requests are still propagating; other parts may be further pruned into smaller trees by other sinks that have initiated acknowledgments. Some restoration mechanism is needed to alert these orphaned nodes and make it possible for them to participate in subsequent augmentation attempts. Step 2 does this restoration as well as the initial repair already described. A node v (necessarily in V) is pruned only if its parent has accepted the acknowledgment of one of v's siblings. This is detectable by v from the state of its parent and causes v to reset. This reset propagates from the root of each pruned piece to its descendants. It is also possible that a reset processor subsequently attaches itself to a new requesting tree before its reset state has propagated to one or more of its subtrees. In this case, such a subtree may simply become part of the new requesting tree, without even discovering that its source has changed.

4.2 The Algorithm

Definitions: The *state* of a processor x is a 4-tuple (pref, aug, choice, dist); *pref* (short for preference) of a processor can be null or p in $N(x)$; *aug*(short for augment) of a processor is s(silent), r(requesting), a (acknowledging) or c(changing); *choice* of a processor can be null or y in $N(x)$; *dist*(short for distance) of a processor can be any non-negative integer less than or equal to $N-2$. If processor x has $x.\text{aug} = r$, then processor x is *requesting*. The terms *silent*, *acknowledging* and *changing* are defined similarly. If a processor x has $x.\text{dist} = d$ then processor *x has distance d*. Throughout this section, a letter (p, y or d) denotes a non-zero or non-null value of a variable. Both zero and null are denoted by 0.

Let $\widehat{N}(x)$ denote the set $N(x) - \{x.\text{pref}\}$. A processor's *state is valid* if it is in state u.1 through u.6 or v.1 through v.5 in Table 1 and its choice is null or in $\widehat{N}(x)$; otherwise it is *invalid*.

Each processor has an arbitrary ordering of its neighbours. For neighbour y of x, $\text{pos}_x(y)$ denotes y's position in x's order. A processor's ordering is only used by itself to break ties between otherwise equivalent neighbours.

Implicit dereferencing of pointers is assumed. For example, if processor x has $x.\text{pref} = p$, then $p.\text{dist}$ refers to the distance field of the processor $y \in N(x)$ that is pointed to by p.

To simplify the description of the algorithm, child and parent relationships are defined for processors that are participating in the search for augmenting paths. Once an augmenting path is fully identified, these relationships are the natural ones along that path. The actual definitions are more complicated because adjacent processors have various combinations of states in the course of an augmentation process.

For a processor $u \in U$ with $u.\text{aug} = r$, let

$$C(u) = \{v \in \widehat{N}(u) \mid v.\text{choice} = u \text{ and } v.\text{aug} = a\}.$$

The *minimum acknowledging child* of processor u, denoted by $\mathbf{mac}(u)$, is defined as

$$\mathbf{mac}(u) = \begin{cases} v & \text{if } v \in C(u) \text{ and } \forall w \in C(u), (v.\text{dist}, \text{pos}_u(v)) \preceq (w.\text{dist}, \text{pos}_u(w)) \\ 0 & \text{otherwise (if } C(u) = \emptyset). \end{cases}$$

(The symbol $\preceq$ denotes lexicographic ordering.)

For a processor $v \in V$ with $v.\text{aug} = s$, let $P(v) = \{u \in \widehat{N}(v) \mid u.\text{aug} = r\}$. The *minimum requesting parent* of processor v, denoted by $\mathbf{mrp}(v)$, is defined as

$$\mathbf{mrp}(v) = \begin{cases} u & \text{if } u \in P(v) \text{ and } \forall w \in P(v), (u.\text{dist}, \text{pos}_v(u)) \preceq (w.\text{dist}, \text{pos}_v(w)) \\ 0 & \text{otherwise (if } P(v) = \emptyset). \end{cases}$$

Let $u \in U$ be a processor with u.aug $= a$, then

$$\textbf{child}(u) = \begin{cases} v & \text{if } u.\text{choice} = v,\ v.\text{choice} = u, \\ & v.\text{aug} = a,\ u.\text{dist} = v.\text{dist} + 1; \\ 0 & \text{otherwise.} \end{cases}$$

Let $v \in V$ be a processor with v.aug $= a$, then

$$\textbf{child}(v) = \begin{cases} u & \text{if } v.\text{pref} = u,\ u.\text{pref} = v, \\ & u.\text{aug} = a,\ v.\text{dist} = u.\text{dist} + 1; \\ 0 & \text{otherwise.} \end{cases}$$

Let $u \in U$ be a processor with u.aug $= r$ or u.aug $= a$, then

$$\textbf{parent}(u) = \begin{cases} v & \text{If } u.\text{pref} = v,\ v.\text{pref} = u, \text{ and} \\ & [(u.\text{aug} = v.\text{aug} = r,\ u.\text{dist} = v.\text{dist} + 1), \text{ or} \\ & (u.\text{aug} = v.\text{aug} = a,\ v.\text{dist} = u.\text{dist} + 1), \text{ or} \\ & (u.\text{aug} = a,\ v.\text{aug} = r)]; \\ 0 & \text{otherwise.} \end{cases}$$

Let $v \in V$ be a processor with v.aug $= r$ or v.aug $= a$, then

$$\textbf{parent}(v) = \begin{cases} u & \text{If } v.\text{choice } = u \text{ and} \\ & [(v.\text{aug} = u.\text{aug} = r,\ v.\text{dist} = u.\text{dist} + 1), \text{ or} \\ & (v.\text{aug} = u.\text{aug} = a,\ u.\text{dist} = v.\text{dist} + 1,\ u.\text{choice } = v), \text{ or} \\ & (v.\text{aug} = a,\ u.\text{aug} = r), \text{ or} \\ & (v.\text{aug} = a,\ u.\text{aug} = c,\ u.\text{pref} = v)]; \\ 0 & \text{otherwise.} \end{cases}$$

Observation 4.1. *If* child$(x) = y$ *for processors* x *and* y, *then* parent$(y) = x$. *If* x *and* y *are both acknowledging processors and* parent$(x) = y$, *then* child$(y) = x$.

Algorithm for processor x: In each composite step, a processor reads all its neighbour's states and then performs the following internal computation.

Step 1 (State validity check): If processor x is in an invalid state, it resets to state u.1 or v.1 depending on its partition.

Step 2 (State consistency check): If the state consistency predicate in Table 1 is false, the processor resets to a new state as given in Table 1.

Step 3 (Preference consistency check): If a processor x is in state u.3, u.4, v.3, v,4 or v.5 and x.pref $= p$ and p.pref $\neq x$ then x resets to state u.1 or v.1 depending on its partition.

Step 4 (State transition): The processor x makes the state transition in Table 2 corresponding to its state if and only if the guard for that state is true.

States for a processor u in partition U:			
No.	State	Consistency predicate	Reset state
u.1	$(0, s, 0, 0)$	true	Not applicable
u.2	$(0, r, 0, 0)$	true	Not applicable
u.3	$(p, s, 0, 0)$	true	Not applicable
u.4	$(p, r, 0, d)$	parent$(u) = p$	$(p, s, 0, 0)$
u.5	(p, a, y, d)	child$(u) = y$ and (parent$(u) = p$ or p.pref $\neq u$)	$(p, s, 0, 0)$
u.6	$(p, c, 0, 0)$	(p.aug $= a$ and p.choice $= u$) or p.pref $= u$	$(p, s, 0, 0)$
States for a processor v in partition V:			
No.	State	Consistency predicate	Reset state
v.1	$(0, s, 0, 0)$	true	Not applicable
v.2	$(0, a, y, 0)$	parent$(v) = y$	$(0, s, 0, 0)$
v.3	$(p, s, 0, 0)$	true	Not applicable
v.4	(p, r, y, d)	parent$(v) = y$	$(p, s, 0, 0)$
v.5	(p, a, y, d)	parent$(v) = y$ and child$(v) = p$	$(p, s, 0, 0)$

Table 1: Consistency predicates for valid states

State transitions for a processor u in partition U:			
No.	Current state	Guard	Next state
1	$(0, s, 0, 0)$	true	$(0, r, 0, 0)$
2	$(0, r, 0, 0)$	mac$(u) = v$ and v.dist $\leq N - 2$	$(v, c, 0, 0)$
3	$(p, s, 0, 0)$	p.aug $= r$ and p.dist $\leq N - 3$	$(p, r, 0, p.\text{dist} + 1)$
4	$(p, r, 0, d)$	mac$(u) = v$ and v.dist $\leq N - 4$	$(p, a, v, v.\text{dist} + 1)$
5	(p, a, y, d)	p.pref $\neq u$	$(y, c, 0, 0)$
6	$(p, c, 0, 0)$	p.pref $= u$	$(p, s, 0, 0)$
State transitions for a processor v in partition V:			
No.	Current state	Guard	Next state
7	$(0, s, 0, 0)$	mrp$(v) = u$ and u.dist $\leq N - 2$	$(0, a, u, 0)$
8	$(0, a, y, 0)$	y.aug $= c$ and y.pref $= v$	$(y, s, 0, 0)$
9	$(p, s, 0, 0)$	mrp$(v) = u$ and u.dist $\leq N - 4$	$(p, r, u, u.\text{dist} + 1)$
10	(p, r, y, d)	p.aug $= a$ and p.dist $\leq N - 3$	$(p, a, y, p.\text{dist} + 1)$
11	(p, a, y, d)	y.aug $= c$ and y.pref $= v$	$(y, s, 0, 0)$

Table 2: State transitions

4.3 Proof of Correctness

The proof of correctness of the protocol is complicated by asynchrony and the need for self-stabilization. What follows is a collection of definitions and lemmas that proves that the algorithm successfully stabilizes to a maximum matching.

This outline begins by establishing a correspondence between a matching in the graph theoretic sense and as defined for the system.

Let $M = \{xy \mid x.\text{pref} = y \text{ and } y.\text{pref} = x\}$.

Lemma 4.1. *The set M is a matching.*

Proof. A processor can have at most one non-null preference at a time. Hence there can never be two adjacent edges in M, and so M is a matching. □

The set M is *the matching of the system*, and the size of M is $|M|$. A processor x is *matched* if $xy \in M$ for some processor y, and x is *matched with* y if $xy \in M$.

For the algorithm to progress, it is essential that processors enter valid states, and Lemma 4.2 ensures that this occurs. It is also important that the size of the matching never decreases. Lemmas 4.3 and 4.4 reflect the behavior of matched processors, and together ensure that the size of M never decreases.

Lemma 4.2. *After the initial round, each processor is always in some valid state.*

Proof. Any processor x is in a valid state immediately after its first execution of Step 1. After any state transition, the next state is always u.1 through u.6 or v.1 through v.5. Processor x always sets its choice to a member in $\widehat{N}(x)$. Whenever x.pref changes, x.choice is set to null. Hence x.choice is always either null or in $\widehat{N}(x)$. □

Lemma 4.2 implies that Step 1 is no-op after the first round. The reminder of the proof applies after the first round.

Observation 4.2. *A processor does not change its preference in Step 2.*

Observation 4.3. *If a processor is matched or its preference is null, then the condition in Step 3 is false and therefore Step 3 is no-op.*

Lemma 4.3. *A U processor does not change its preference while it is matched.*

Proof. By Observations 4.2 and 4.3, the only way a matched U processor can change its preference is in Step 4. A non-null preference changes in Step 4 only in line 5 of Table 2, when a processor is not matched. □

Lemma 4.4. *A matched V processor never becomes unmatched.*

Proof. By Lemma 4.3, a U processor never changes its preference while matched. Hence the only way a matched V processor may become unmatched is if it changes its own preference. Because no processor changes its preference in Step 2, and Step 3 is no-op for a matched processor, the only way a matched V processor can change its preference is in Step 4, by executing line 11 of Table 2. In this case it is immediately matched with its choice. □

Corollary 4.1. *The cardinality of M never decreases.*

Progress is made through requests and acknowledgments attempting to find alternating paths from unmatched U processors to unmatched V processors. Finding these paths is crucial for such progress to occur.

The acknowledgment of a processor x is *real* if either x is in partition V and in state v.2 (processor x is called a *sink*), or child$(x) = y$ and y's acknowledgment is real. If a processor's acknowledgment is not real then it is *spurious*. Informally, a *real acknowledgment* refers to an acknowledging processor that is on an alternating path that is reserved all the way to an unmatched V processor (a *sink*). If x_1 is an acknowledging processor and it's acknowledgment is real, then by definition either x_1 is a sink or there exists a path $x_1, \ldots, x_m$ such that child$(x_i) = x_{i+1}, 1 \leq i \leq m-1$, and x_m is a sink. This path is called an *acknowledging path*. Observe that by definition of child, processors $x_2, \ldots, x_{m-1}$, are matched.

Lemma 4.5. *If $x_1, \ldots, x_m$ is an acknowledging path leading to sink x_m, then no processor on this path can change state before x_1 changes.*

Proof. Each $x_i, 2 \leq i \leq m-1$, is either in state u.5 or v.5. By definition of a real acknowledgment, each has a non-zero child and by Observation 4.1 each also has a non-zero parent. Processor x_m is in state v.2 and has a non-zero parent. Therefore the state consistency predicate is true and Step 2 is a null step for $x_i, 2 \leq i \leq m$.

Since x_m has zero preference and $x_i, 2 \leq i \leq m-1$, is matched, Step 3 is a null step for $x_i, 2 \leq i \leq m$, by Observation 4.3.

Each U processor in the path $x_i, 2 \leq i \leq m-1$, is in state u.5 and is matched. So its guard in line 5 is false. Hence no U processor in this path can make a state change before x_1.

Any V processor, v, in the acknowledging path is either in state v.2 (processor x_m) or in state v.5 (any other V processor). Let parent$(v) = v$.choice $= u$. Then u must be a U processor in the acknowledging path. For the guards in line 8 (for processor x_m) or line 11 (for any other V processor) to be true, u.aug $= c$. But no U processor in the acknowledging path has changed state. So the guards in line 8 and line 11 are false. Hence no V processor in the acknowledging path can change state. □

Corollary 4.2. *After the initial round, a real acknowledgment never becomes spurious.*

Proof. By Lemma 4.2 any processor v in V with v.pref $= 0$ and v.aug $= a$ must be in state v.2, that is by definition real.

Let x_1 be real acknowledging processor with non-zero preference. Then there exists an acknowledging path $x_1, \ldots, x_m$ (to a sink), and no processor on this path can change state before x_1. Hence x_1's acknowledgment will never become spurious. □

Lemma 4.6. *After N rounds, no processor will have a spurious acknowledgment.*

Proof. The proof proceeds by induction on the value of the distance field of an acknowledging processor.

By Lemma 4.2, after the initial round, any acknowledging processor with zero distance must be in state v.2, that is by definition real.

Suppose that after $k \geq 1$ rounds, any acknowledging processor x with x.dist $\leq k-1$ has a real acknowledgment. Let x be an acknowledging processor with x.dist $= k$ at the start or round $(k+1)$. By definition, child$(x) = 0$ or child$(x) = y$ for some acknowledging processor y with y.dist $= k-1$. If child$(x) = 0$, then the state consistency predicate (states u.5 or v.5 in Table 1) for x is false, triggering a reset to x.aug $= s$. Otherwise, by the induction hypothesis y's acknowledgment is real. Therefore x has a real acknowledgment. Hence at the end of round $(k+1)$, x has a real acknowledgment or is no longer acknowledging. Also during round $(k+1)$, if a processor x becomes acknowledging with x.dist $= l \leq k$, then child(x) has distance $\leq k-1$, which ensures that child(x) has a real acknowledgment. Thus x has a real acknowledgment.

Since no processor copies an acknowledgment with distance $\geq N-2$, the lemma follows. □

The identification of potential augmenting paths is achieved through the parent relationship between processors that are requesting or acknowledging. The *root* of such a path is a processor with no parent. An augmenting path is found when the root of a sink is a *source* (an unmatched U processor).

Definition 4.1. *Let x be a requesting or acknowledging processor. If* parent$(x) = 0$*, then* $\mathbf{root}(x) = x$*. If* parent$(x) \neq 0$*, then there exist processors $x_1, \ldots, x_m$ such that* parent$(x) = x_1$, parent$(x_i) = x_{i+1}, 1 \leq i \leq m-1$*, and* parent$(x_m) = 0$*. In this case* $\mathbf{root}(x) = x_m$*.*

Definition 4.2. *A processor $u \in U$ is a* source *if u is in state u.2, or u is in state u.6, or u is in state u.5 with u.pref $= v$ and v.pref $\neq u$.*

Lemma 4.7. *If u is a source and v is a sink such that* root$(v) = u$*, then the vu-path $Q = v, x_1, \ldots, x_m, u$ with* parent$(v) = x_1$, parent$(x_i) = x_{i+1}, 1 \leq i \leq m-1$*, and* parent$(x_m) = u$ *is an augmenting path based on M. Furthermore, the state consistency predicate is true for any processor on this path.*

Proof. If $m = 0$, then u and v are two adjacent free processors. Hence u, v is an augmenting path of length one.

If $m \geq 1$, then rewrite the path as $Q = v, u_1, v_1, u_2, v_2, \ldots, u_l, v_l, u$, where $l = m/2$. Then parent$(u_i) = v_i, 1 \leq i \leq l$. By definition of parent of a U processor $u_i v_i \in M$. Hence Q is an augmenting path.

Every processor x on Q, except for the source, has parent$(x) \neq 0$. If x and y are two processors on Q such that parent$(x) = y$ and y.aug $= a$, then by definition of parent, x.aug $= a$ and by Observation 1, child$(y) = x$. Hence the state consistency predicates are true for all processors on Q. □

Definition 4.3. *Let x_1 be a processor with x_1.aug $= r$ and* root$(x_1) = x_m$*, where x_m is in state u.2. Then there exists a path $x_1, \ldots, x_m$, such that* parent$(x_i) = x_{i+1}, 1 \leq i \leq m-1$*. This path is called a* requesting path.

Observe that by definition of parent, x_i.aug $= r, 1 \leq i \leq m$. Also processor $x_i, 2 \leq i \leq m-1$, is matched.

Definition 4.4. *Let* $Q = v, x_1, \ldots, x_m, u$ *with* $\text{parent}(v) = x_1$, $\text{parent}(x_i) = x_{i+1}, 1 \le i \le m-1$, *and* $\text{parent}(x_m) = u$ *be an augmenting path between sink* v *and source* u. *Then* Q *is a partially acknowledged augmenting path or simply a* PA-path. *If* $u.\text{aug} \neq r$ *then* Q *is called a fully acknowledged augmenting path or an* FA-path.

If there exist processors x_i *and* x_{i+1} *on this PA-path such that* $x_i.\text{aug} = a$ *and* $x_{i+1}.\text{aug} = r$, *then the path* $x_{i+1}, \ldots, x_m, u$ *is a requesting path. The* size of the unacknowledged part of the PA-path *is defined as the number of processors on the path* $x_{i+1}, \ldots, x_m, u$.

An augmenting path that is identified through the parent pointers can be either *fully acknowledged* (*FA-path*) or *partially acknowledged* (*PA-path*), depending on whether it is reserved all the way to the source or not. Partially acknowledged augmenting paths are the key to progress, as outlined in Lemmas 4.8 and 4.9.

Lemma 4.8. *After* N *rounds, if there is a PA-path such that the size of it's unacknowledged part is* k, *then within* k *rounds there will be an FA-path.*

Proof. The proof proceeds by induction on the size of the unacknowledged part of a PA-path.

Suppose there is a PA-path Q whose unacknowledged part has size one. Then

$$Q = u, x_1, \ldots, x_m, v$$

where u is a source with $u.\text{aug} = r$, and $x_1, \ldots, x_m, v$ is an acknowledging path with $\text{parent}(x_1) = u$. Hence the guard in line 2 of Table 2 is true for u, and after u executes line 2, Q will be an FA-path.

Assume that the result is true for some $k \ge 1$ and suppose that $R = u, x_1, \ldots, x_m, v$ is a PA-path with source u whose unacknowledged part has size $(k+1)$; i.e, $u.\text{aug} = r$ and $x_i.\text{aug} = r, 1 \le i \le k$. By Lemma 4.5 the only processor that can change state on the path $x_{k+1}, \ldots, x_m, v$ is x_{k+1}. But x_{k+1} cannot change state before $\text{parent}(x_{k+1}) = x_k$ changes state.

By Lemma 4.7, R is an augmenting path, implying that $x_{2i+1} \in V, 0 \le i \le m/2 - 1$, and $x_{2i+1}x_{2(i+1)} \in M$. The only possible state change for a requesting V processor x_i on R is line 10. But this requires $x_{i+1}.\text{aug} = a$, which is only possible if $i = k$. If x_k changes state, the path R will be a PA-path whose unacknowledged part has size k.

The possible state changes for a requesting U processor on R are line 2, for source u, and line 4, for any other processor x_i. In either case there exists a V processor v with a real acknowledgment such that $\text{parent}(v) = u$ or $\text{parent}(v) = x_i$. Since v has a real acknowledgment (Lemma 4.6), there is an acknowledging path $v, y_1, \ldots, y_l, l \ge 1$. If u executes line 2, we have an FA-path $u, v, y_1, \ldots, y_l$. Otherwise, if x_i executes line 4, there is a PA-path $u, x_1, \ldots, x_i, v, y_1, \ldots, y_l$ whose unacknowledged part has size at most k. Hence the result follows by induction. □

Lemma 4.9. *After* N *rounds, if there is an FA-path, then the size of the matching will increase within* n *rounds.*

Proof. The proof proceeds by induction on the length of the FA-path.

Suppose u, v is an FA-path of length one. If $u.\text{aug} = a$ then the guard in line 8 is false for v and guard in line 5 is true for u. Hence after the first round, $u.\text{aug} = c$ making the guard in line 8 is true for v. During the second round, v will execute line 8, and u and v will be matched with each other. As they were unmatched before, the cardinality of the matching increases by one.

Assume that the result is true for all FA-paths with length k or less and let $u, x_0, ..x_k$ be a FA-path of length $k+1$ with source u. If $u.\text{aug} = a$, then by definition of source, the guard in line 5 will be true for u, and in the next round, $u.\text{aug} = c$. If $u.\text{aug} = c$ then, since $\text{parent}(x_0) = u$, the guard in line 11 is true for x_0. After x_0 takes that step, x_1 becomes the new source. Hence there is an FA-path of length $k-2$ within two rounds. The result follows by induction. □

Thus the size of M will increase if there is a PA-path, and certain changes of the configuration of the system guarantee the existence of a PA-path. One such change is when an acknowledging processor becomes an *orphan*, a consequence of the pruning process. A second change that ensures the existence of a PA-path is the creation of a *spurious request*, another consequence of the pruning process.

Definition 4.5. *An acknowledging processor x is an* orphan *if its acknowledgment is real,* $\text{root}(x) = y$ *and y is a not a source.*

Lemma 4.10. *After N rounds, if an acknowledging processor becomes an orphan, then there exists a PA-path.*

Proof. Let x be a non-orphaned processor with a real acknowledgment. Then there exists a path

$$Q = x_1, x_2, \dots, x_{l-1}, x_l = x, x_{l+1}, \dots, x_m$$

where x_m is a sink and $\text{root}(x_l) = x_1$ is a source. If x_l becomes an orphan, then some processor $x_i, 1 \le i \le l-1$, changes state. We consider three cases depending on the state of the source x_1.

If x_1 is in state u.6, then x_2 must be acknowledging. Because $x_2, \dots, x_m$ is an acknowledging path to a sink x_m, Lemma 4.5 guarantees that the only processor on this path that can change state is x_2. The only possible state change for x_2 is line 11, making x_3 a source. Thus x_l does not become an orphan.

If x_1 is in state u.5, then $x_1, \dots, x_m$ is an acknowledging path to a sink x_m. Again, Lemma 4.5 ensures that the only processor on this path that can change state is x_1. The only possible state change for x_1 is line 5. But this does not change $\text{parent}(x_2)$, and x_1 remains a source. Hence x_l does not become an orphan.

Thus, if x_l becomes an orphan, x_1 must be in state u.2. This implies that for some $k, 1 \le k \le l-1$, $x_1, x_2, \dots, x_k$ is a requesting path, and $x_{k+1}.\text{aug} = a$. Since $x_{k+1}, \dots, x_m$ is an acknowledging path to a sink x_m, Lemma 4.5 ensures that only processor on this path that can change state is x_{k+1}. However, x_{k+1} cannot change state before its parent, x_k changes state.

If $x_k \in V$, then the only possible state change is line 10 of Table 2. This state change does not alter the parent value of any processor on Q. If $x_k \in U$, then the possible state changes are line 2 or line 4 of Table 2. In either case the execution of these lines requires a processor $v \in V$ with $\text{parent}(v) = x_k$. Since v's acknowledgment is real (Lemma 4.6), either

v is a sink or there exists an acknowledging path $v, y_1, \ldots, y_t$ ending at sink y_t. In the first case $x_1, \ldots, x_k, v$ is a PA-path. In the second case $x_1, \ldots, x_k, v, y_1, \ldots, y_t$ is a PA-path. □

Definition 4.6. *The request of a processor x is* real *if* $\text{root}(x) = y$ *and y is in state u.2. If the request of a processor is not real, then it is* spurious.

Lemma 4.11. *After N rounds, if the request of a processor changes from real to spurious, then there exists a PA-path.*

Proof. By Lemma 4.2, after the first round, any requesting processor is in state u.2, u.4 or v.4. If processor is in state u.2, then it is real and cannot be spurious.

Suppose processor x_1 is in state u.4 or v.4 and has a real request. Then by definition there exists a requesting path $x_1, \ldots, x_m$ ending at source x_m. Suppose x_1's request becomes spurious because of the state change of some processor x_i on this path with $\text{parent}(x_{i-1}) = x_i$, $i > 1$. If x_i is in partition V, then by definition of parent of a U processor, $x_i.\text{pref} = x_{i-1}$. The only state change possible for x_i (state v.4) is line 10, and this requires $x_{i-1}.\text{aug} = a$. But $x_{i-1}.\text{aug} = r$. Hence x_i can not be in partition V. The possible state changes for a requesting U processor are either line 2 for processor x_m in state u.2, or line 4 for processor x_i ($i < m$) in state u.4. In either case, there exists an acknowledging V processor v with $\text{parent}(v) = x_i$. Since v's acknowledgment is real (Lemma 4.6), either v is a sink or there exists an acknowledging path $v, y_1, \ldots, y_l$ ending at sink y_l. In the first case $x_m, \ldots, x_i, v$ is a PA-path. In the second case $x_m, \ldots, x_i, v, y_1, \ldots, y_l$ is a PA-path. □

If no spurious requests are created for an appropriate number of rounds, then Lemma 4.12 implies that the spurious requests all disappear. At this point, either a PA-path is again ensured, or else the system also has no orphaned acknowledging processor (Lemma 4.13). In the case that the system has no orphaned acknowledgments and no spurious requests, some augmenting path will be identified (as established in the proof of Lemma 4.13) as long as M is not a maximum matching (Lemma 4.14). Once a maximum matching is attained, Lemma 4.15 shows that stability is achieved.

Lemma 4.12. *If no processor's request changes from real to spurious in rounds r to $(r + l)$ inclusive, $r > N, l > N$, then there will be no spurious request in rounds $(r + N)$ to $(r + l)$ inclusive.*

Proof. It suffices to prove that all the spurious requests in the system at the start of round r will disappear after the end of round $(r + N - 1)$.

The proof proceeds by induction on the value of the distance field of a requesting processor.

By Lemma 4.2, after round r, any requesting processor with zero distance must be in state u.2, which is by definition real.

Suppose that after $r + k$ ($k \geq 0$) rounds any requesting processor x with $x.\text{dist} \leq k$ has a real request. Let x be a requesting processor with $x.\text{dist} = k + 1$, at the start or round $(r + k + 1)$. By definition, $\text{parent}(x) = 0$, or $\text{parent}(x) = y$ for some requesting processor y with $y.\text{dist} = k$. If $\text{parent}(x) = 0$, then the state consistency predicate (state u.4 or v.4 in Table 1) for x is false, triggering a reset to $x.\text{aug} = s$. Otherwise, by the induction hypothesis, y's request is real. Therefore x has a real request. Hence at the end of round

$(r+k+1)$, x has a real request or is no longer requesting. Also during round $(r+k+1)$, if a processor x becomes requesting with x.dist $= l \leq (k+1)$, then parent(x) has distance $\leq k$, which ensures that parent(x) has a real request. Thus x has a real request.

Because no processor copies a request with distance $\geq N-2$, the lemma follows. □

Lemma 4.13. *After N rounds, if there is no spurious request for n consecutive rounds, then either all orphaned processors become silent or there exists a PA-path.*

Proof. Suppose x_i is an orphaned processor, with root$(x_i) = x_1$. By definition, x_1 is not a source. Since there is no spurious request, x_1.aug $= a$. Also since x_i is real there exists an acknowledging path $x_1, \ldots, x_i, \ldots, x_m$ ending at sink x_m. Hence $x_1, \ldots, x_m$ is an acknowledging path of length $m-1$ and all processors on this path are orphans. Since there can be no path of length greater than $n-1$, it suffices to show that after one round, either the length of this path decreases or there exists a PA-path.

Since parent$(x_1) = 0$ and x_1 is not a source, the state consistency predicate is false for x_1. Hence after the first round, either parent$(x_1) = y$ for some requesting processor y, in which case we have a PA-path, or x_1 becomes silent, in which case the length of the acknowledging path decreases. Thus after n rounds, either all orphaned processors become silent or there exists a PA-path. □

Lemma 4.14. *After N rounds, if the matching is not maximum, then its cardinality will increase within $N+4n$ rounds.*

Proof. Consider an interval I of $N+2n$ rounds beginning at round $r > N$.

If during interval I, the request of a processor changes from real to spurious, or any acknowledging processor becomes orphaned, then by Lemma 4.11 or Lemma 4.10 respectively, there is a PA-path.

Otherwise, during I, no processor's request changes from real to spurious and no processor becomes orphaned. By Lemma 4.12, after round $(r+N-1)$, there are no spurious requests. Thus by Lemma 4.13, either there will be a PA-path, or there are no orphaned acknowledging processors after round $(r+N+n-1)$.

In the second case, since the matching is not maximum, there is an augmenting path. An easy induction on the length of this path shows that, if there is no spurious request or orphaned acknowledging processor, a request will propagate from a source to a sink giving a PA-path within n rounds. Hence before the end of round $(r+N+2n-1)$, there is a PA-path.

In all cases there is a PA-path before the end of round $(r+N+2n-1)$. It now follows from Lemma 4.8 and Lemma 4.9 that the cardinality of M will increase within another $2n$ rounds. □

Lemma 4.15. *After N rounds, if the matching is maximum, then no matched processor changes its preference.*

Proof. Suppose M is a maximum matching and $uv \in M$. By Lemma 4.3, u.pref does not change as long as $uv \in M$ (i.e. v.pref $= u$). If v.pref changes, then it must be by line 11. In this case v is in state v.5 and by Lemma 4.6 its acknowledgment is real. Thus there is an acknowledging path $v, x_1, \ldots, x_m$, where x_m is a sink. Also, there must be a processor u_0 in state u.6, such that v.choice $= u_0$ and u_0.pref $= v$ (refer to the guard in line 11). Hence parent$(v) = u_0$, and therefore $u_0, v, x_1, \ldots, x_m$ is an FA-path. By Lemma 4.7, this is an augmenting path, contradicting the assumption that M is maximum. Therefore the matching does not change. □

Lemmas 4.14 and 4.15 and Corollary 4.1 combine to prove the main result.

Theorem 4.2. *Starting from an arbitrary configuration, the system will stabilize to a maximum matching within $O(Nn)$ rounds.*

5 A 2-colouring Algorithm for Bipartite Graphs

The algorithm presented in this section is motivated by the leader election algorithm in [4, pp. 34-36].

Model: The model of computation is locally shared memory, under read/write atomicity and a distributed scheduler. All processors have distinct id's. Every processor has an upper bound N for the number of processors in the system.

Overview: The basic idea is to propagate the colour and id of the processor with minimum id. To do so, every processor maintains, along with its colour, an origin field containing the id of the source of the colour (i.e. the id of the processor that was used to choose a colour). Because this is a self stabilizing algorithm, the system might start in a configuration with spurious origins. One way to get rid of these is to introduce a distance field that keeps track of the distance of a processor from its origin. When the processor with minimum id sets its colour to U and sets its origin to itself it also sets its distance to 0. When a processor gets its colour by complementing the color of its neighbour, it increases the distance by one. This means that the distance of a spurious origin keeps increasing as it gets copied until it reaches N. Eventually all the spurious origins are deleted and the system converges to a 2-colouring within $D + 1$ rounds, where D is the diameter of the graph.

Definitions and data structures: The state of a processor is a 4-tuple (id, colour, origin, distance); id is the read-only identifier of a processor and can be any positive integer; colour is either U or V; orig (short form of origin) of a processor is null (denoted 0), or an id of a processor; dist (short form of distance) is the distance of the processor from the origin and can be any non-negative integer less than N.

Since the objective is to 2-colour, the complements of colours U and V are denoted as $\overline{U} = V$ and $\overline{V} = U$ respectively.

For processor x, let $C(x) = \{y \in N(x) \mid y.\text{orig} \neq 0 \text{ and } y.\text{orig} < x.\text{id and } y.\text{dist} \leq N-2\}$. Define the *minimum origin of* x, denoted $m(x)$, as

$$m(x) = \begin{cases} y & \text{if } y \in C(x) \text{ and } \forall z \in C(x), (y.\text{orig}, y.\text{id}) \preceq (z.\text{orig}, z.\text{id}) \\ 0 & \text{if } C(x) = \emptyset \end{cases}$$

(with $\preceq$ denoting lexicographic order).

Algorithm for processor x:

```
do forever
   if (m(x) = 0)
      x.colour ← U
      x.orig ← x.id
      x.dist ← 0
   else
      x.colour ← ‾m(x).colour‾
      x.orig ← m(x).orig
      x.dist ← m(x).dist + 1
```

Proof of correctness:

Definition 5.1. *An origin of a processor x is* real *if x.orig $= 0$ or there exists a processor y, such that x.orig $= y$.id. If the origin of a processor is not real, then it is* spurious.

Lemma 5.1. *After N rounds there will be no spurious origin in the system.*

Proof. Note that in every step, a processor always sets its variables. The proof proceeds by induction on the value of the distance field.

Whenever a processor x sets its distance field to 0, it sets x.orig $= x$.id. Hence after the first round, the origin of every processor x, with x.dist $= 0$, is real.

Assume that after $l \geq 1$ rounds, there is no spurious origin with distance less than l. During round $(l + 1)$, if processor x sets x.dist $= m \leq l$, then $m(x)$.dist $\leq l - 1$. By the induction hypothesis $m(x)$.orig is real. Hence after round $(l+1)$, there is no spurious origin with distance less than $(l + 1)$. Since no processor copies an origin with distance more than $N - 2$, the result follows. □

Lemma 5.2. *If the system reaches a configuration with no spurious origin, then the system will be 2-coloured within $D + 1$ rounds.*

Proof. Suppose that the system has no spurious origins and x be the processor with the minimum id in the system. After the first round, x.colour is correct.

A processor y is *properly coloured* with respect to processor x if y.orig $= x$.id and

$$y.\text{colour} = \begin{cases} x.\text{colour} & \text{when } d(x,y) \text{ is even} \\ \\ \overline{x.\text{colour}} & \text{when } d(x,y) \text{ is odd} \end{cases}$$

Let y be a processor with $d(x, y) = k \geq 1$. A straightforward induction on k ensures that y is properly colored with respect x after $k + 1$ rounds, and the result follows. □

Theorem 5.1. *Starting from an illegitimate colouring, the system will reach a proper 2-colouring within $N + D + 1$ rounds.*

Proof. The proof follows immediately from Lemma 5.1 and 5.2. □

6 Comments and Future Work

For the sake of clarity, we assumed locally shared memory for the maximal and the maximum matching algorithms. However, it can be easily proved that both algorithms are correct for the link-register model as well.

Both the maximal matching and the maximum matching algorithms are dynamic, meaning that if the network size and topology change, the system will re-converge to a legitimate configuration. In case of the maximum matching algorithm, the new system size can not exceed the upper bound on the system size.

In the maximum matching algorithm it is assumed that the processors know an upper bound on the number of processors. This upper bound in conjunction with the distance field allows the processors to get rid of the spurious acknowledgments and requests. The distance field also helps to avoid a cycle of processors being stuck in requesting state where each processor's parent is its left neighbour, there by blocking a genuine request from propagating. It can be shown that the upper bound on the number of processors is not necessary for progress. However in that case we need unbounded counter for the distance field which is not feasible in a self-stabilizing system. Any upper bound on the distance field implicitly imposes an upper bound on the number of processors, which is exponential on the length of the distance field. This bound appears in the current complexity measure, which makes the algorithm to have exponential complexity. Our guess is that we can prove that the algorithm converges on $O(n^2)$ rounds. However the proof will be more involved and complicated.

The proof of the maximum matching algorithm assumes composite atomicity as opposed to the more realistic read/write atomicity. Also the maximum matching algorithm only works for bipartite graphs. In a general graph, the presence of odd cycles makes it difficult to find augmenting paths. There is a simple technique [12, pp. 226-243] to overcome this problem in a sequential algorithm. Implementing that technique in a distributed setting poses challenges. Our future work will be directed towards solving these problems.

References

[1] J. Beauquier, A.K. Datta, M. Gradinariu, and F. Magniette. Self-stabilizing local mutual exclusion and daemon refinement. *14th International Symposium on Distributed Computing, DISC00*, pages 223–237, 2000.

[2] C. Berge. Two theorems in graph theory. *Proc. Nat. Acad. Sci. U.S.A.*, 43:842–844, 1957.

[3] E.W. Dijkstra. Self-stabilization in spite of distributed control. In *Selected Writings on Computing: A Personal Perspective*, pages 41–46. Springer-Verlag, 1982. Original year of publication of the paper is 1973.

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

[5] S. Dolev, A. Israeli, and S. Moran. Self stabilization of dynamic systems. In *Proceedings of the MCC Workshop on Self-Stabilizing Systems, MCC Technical Report No. STP-379-89*, 1989.

[6] J. Edmonds. Paths, trees and flowers. *Canad. J. Math*, pages 449–67, 1965.

[7] M. Gardinariu and C. Johnen. Self-stabilizing neighbourhood unique naming under unfair schedular. In *Euro-Par 2001*, volume 2150, pages 458–465. Springer-Verlag, 2001.

[8] S.T. Hedetniemi, D.P. Jacobs, and P.K. Srimani. Maximal matching stabilizes in time O(m). *IPL*, 80:221–223, 2001.

[9] J.E. Hopcroft and R.M. Karp. A $n^{5/2}$ algorithm for maximum matching in bipartite graphs. *SIAM J. on Computing*, 2:225–31, 1973.

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

[11] M.H. Karaata and K.A. Saleh. A distributed self-stabilizing algorithm for finding maximum matching. *Computer Systems Science and Engineering*, 3:175–180, 2000.

[12] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, 1982.

[13] S. Sur and P.K. Srimani. A self-stabilizing algorithm for coloring bipartite graphs. *Information Sciences*, 69:219–227, 1993.

[14] G. Tel. Maximal matching stabilizes in quadratic time. *IPL*, 49:271–272, 1994.

[15] D.B. West. *Introduction to Graph Theory.* Prentice-Hall, 2001.