Dynamic and Self-stabilizing Distributed Matching

Subhendu Chattopadhyay, Lisa Higham, Karen Seyffarth

January 26, 2003

Abstract

Self-stabilization is a unified model of fault tolerance. A self-stabilizing system can recover from an arbitrary transient fault without re-initialization. Self-stabilization is a particularly valuable attribute of distributed systems since they are typically prone to various faults and dynamic changes.

In several distributed applications, pairing of processors connected in a network can be viewed as a matching of the underlying graph of the network. A self-stabilizing matching algorithm can be used to build fault tolerant paring of clients and servers connected in a network.

First contribution of this report is an efficient, dynamic and self-stabilizing maximal matching algorithm for arbitrary anonymous networks. The algorithm implements a locally distinct label generation technique that can be used by other applications. The second contribution of this report is a dynamic and self-stabilizing maximum matching algorithm for arbitrary bipartite networks. This is the first distributed maximum matching algorithm for networks containing cycles.

Chapter 1

Introduction

A distributed system is a collection of processes, connected by some interconnection mechanism, that communicate among themselves to solve a common problem. The processing units can run in the same machine and communicate by interprocess communication such as message queues or shared memory, or they can run in different computers and communicate by a network. Distributed algorithms run on distributed systems to solve problems that are inherently distributed in nature. They include problems of resource allocation, distributed consensus, symmetry breaking and deadlock prevention. Unlike parallel algorithms, generally distributed algorithms are not meant to reduce the computing time of a sequential algorithm.

Often a distributed system is modelled as a graph where nodes represent processes and edges represent the communication topology. When modelled this way, various distributed problems can be viewed as graph theoretic problems in the underlying graph. For example, if a distributed system consists of a network of clients and servers and we want to pair up clients with servers, the problem can be viewed as a matching problem in the underlying graph of the network. This pairing would be optimal if the matching is a maximum matching in the graph. For another example, a distributed system may have costs associated with the communication channels and we may want to determine the minimum cost broadcast route in that network. This problem can be viewed as finding the minimum spanning tree in the underlying graph where each edge has weight proportional to the cost of communication.

In a distributed system, each process can be modelled as a state machine and the state of a process is the values of its variables. The collection of the states of all the processes determines the system configuration. A nonempty subset of the set of configurations is defined as the initial configurations. Another nonempty subset of the set of configurations is defined as the final or legitimate configurations. The system starts in some initial configuration. Then, the processes communicate with each other. Based on the information they receive, they change states and the system enters a new configuration. Depending on the system, all processes may proceed with their computation at the same speed, or at different speeds. After repeating this process for a bounded amount of time, the system reaches some predetermined final configuration and remains in that configuration thereafter. At this time, the output of the system is manifested by the states of the processes.

A distributed system consists of many processes and communication channels. The probability that one or more components may fail increases with the number of components. Therefore it is desirable to design distributed systems that can withstand faults. For example, consider again a distributed system that solves the maximum matching problem. After the system finds the maximum matching, the matching will be represented by the states of the individual processes. If the state of a matched process becomes corrupted, then the system configuration may no longer be a maximum matching.

Another problem with distributed systems is frequent changes in the network topology. In our example of the distributed system consisting of clients and servers, the clients may exit the system once they are serviced, or new servers can be added to the system. This changes the underlying graph of the network and may render the matching between clients and servers invalid.

Once a fault or a topology change takes the system to an illegitimate configuration, one way to fix this is to shut down the whole system, reinitialize it and then restart it. But this can be expensive and in some cases it may even be infeasible. It would be more useful if algorithms are designed that do not need initialization, meaning that no matter what configuration the system starts from, after a finite amount of computation the system is guaranteed to reach a legitimate configuration. This approach is called self-stabilization. In a self-stabilizing system, after the system reaches a legitimate configuration, a transient fault may move the system to an illegitimate configuration. But because of the self-stabilizing property, after a finite amount of computation the system again reaches a final configuration, without requiring external intervention. If a self-stabilizing system can also recover from a network topology change, it is called a dynamic system.

The first contribution of this report is a dynamic and self-stabilizing maximal matching algorithm for arbitrary networks. The algorithm is efficient, works on a realistic model of computation and the processes do not need identifiers. The algorithm works in two phases. In the first phase, locally distinct labels are generated using randomization. These labels are used in the second phase to determine a maximal matching. The combined algorithm has expected time complexity that is logarithmic in the number of processes for most networks. The locally distinct labels can be used for other applications.

The second contribution of this report is a dynamic and self-stabilizing maximum matching algorithm for arbitrary bipartite networks. To the best of our knowledge, this is the first distributed maximum matching algorithm for networks containing cycles. This algorithm also works in two phases. In the first phase, the bipartite network is bipartitioned using a 2-coloring algorithm. This partition information is then used in the second phase to find a maximum matching. The combined algorithm has time complexity that is quadratic in the upper bound on the number of processes.

These algorithms can be used in a variety of applications. For example, in a distributed system consisting of clients and servers, these algorithms can be used to pair up clients and servers optimally. Since the algorithms are self-stabilizing, the system can recover from transient faults that might corrupt the processes states. Furthermore, since the algorithms are dynamic, if the servers are stateless, the system can automatically readjust to a maximal or a maximum matching once a client is serviced and is removed from the system.

Outline of the report Chapter 2 contains necessary background and definitions. Related previous work is described in Chapter 3. Chapter 4 contains the maximal matching algorithm and a randomized algorithm for distributing locally distinct labels. Chapter 5 contains the 2-coloring algorithm and the maximum matching algorithm. Conclusions and future work are presented in Chapter 6.

Chapter 2

Graph Theory and Distributed Systems

2.1 Graph Theory

2.1.1 Introduction

The following definitions and results can be found in most introductory graph theory texts (for example [21]).

A *graph* G consists of a nonempty set V of *nodes*, a set E of *edges*, and an *incidence* relation that associates each edge with two nodes. If $x, y \in V$ are two nodes in a graph, then an edge incident between them is denoted as xy. If there is an edge xy between nodes x and y, then x and y are *adjacent nodes*. If two edges are incident to the same node, then they are *adjacent edges*. An edge with identical ends is a *loop*. If two or more edges are incident with the same pair of nodes, then they are *multiple* edges. A graph is *simple* if it does not contain loops or multiple edges. In this report we will consider only simple graphs. A graph is *complete* if there is an edge between every pair of nodes.

A *path* in a graph is a sequence of nodes $v_1, \ldots, v_m$, where $v_i \in V, 1 \leq i \leq m,\ v_i v_{i+1} \in E, 1 \leq i < m$ and $v_i \neq v_j, 1 \leq i < j \leq m$. A *cycle* in a graph is a sequence of nodes $v_1, \ldots, v_m$, where $v_i \in V, 1 \leq i \leq m,\ v_i v_{(i+1) \bmod m} \in E, 1 \leq i \leq m$ and $v_i \neq v_j, 1 \leq i < j \leq m$. The *length* of a path or a cycle is the number of edges in it. A cycle is an *odd cycle* if it has an odd number of edges. The *distance* between two nodes $x, y \in V$ in a graph, denoted $d(x, y)$, is the length of the shortest path between the nodes. For the sake of completeness, $d(x,x)$ is defined as 0. The *diameter* of a graph, denoted D is,

$$D = max\{d(x,y) \mid x,y \in V\}.$$

A graph is *connected* if there is a path between any two nodes in the graph.

The *neighbourhood* of a node $x \in V$ in a graph, denoted $N(x)$, is the set of nodes adjacent to x. Formally

$$N(x) = \{y \in V | xy \in E\}.$$

For any finite set S, the *cardinality* or the *size* of S, denoted as $|S|$, is the number of elements in S. The *degree* of a node x, denoted $\delta(x)$, is $\delta(x) = |N(x)|$. The maximum degree of a graph, denoted Δ, is

$$\Delta = max\{\delta(x) | x \in V\}.$$

The *neighbourhood within distance two* of a node $x \in V$ in a graph, denoted $N_2(x)$, is

$$N_2(x) = \{y \in V | 1 \leq d(x,y) \leq 2\}.$$

The *closed neighbourhood* of a node x, denoted $N[x]$, is $N(x) \cup \{x\}$.

A graph is *bipartite* if its node set can be partitioned into two sets U and V, such that every edge is incident to one node from U and another node from V. Partitioning the node set of a bipartite graph in this fashion is referred to as *2-coloring the bipartite graph.* The following theorem gives an important property of bipartite graphs.

Theorem 2.1.1 (König, 1936). *A graph is bipartite if and only if it does not contain any odd cycle.*

A connected graph is a *tree*, if it does not contain any cycle. Note that a tree with n nodes has $n-1$ edges. A node x in a tree is a *leaf* if $|\delta(x)| = 1$.

2.1.2 Matching

Let G be a graph with node set V and edge set E. A *matching* in G is a subset M of E in which no pair of edges are 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. An *alternating path in G, with respect to 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. The following well known theorem by Berge gives not only an interesting property of maximum matching but also leads to an algorithm to find one.

Theorem 2.1.2 (Berge, 1957). *A matching M in a graph G is maximum if and only if there is no augmenting path in G with respect to M.*

Proof. Sufficiency: Let M be a matching and P be an augmenting path with respect to M. Create M' from M, by excluding the matched edges in P, and including the unmatched edges of P and all other edges of M that are not on P. Then M' is a matching. But the size of M' is one bigger than that of M. Hence M is not a maximum matching.
Necessity: Let M be a matching that is not maximum. Let M' be a maximum matching. Consider the subgraph G' of G consisting of its original node set and the edge set $(M' - M) \cup (M - M')$. Since M is a matching, there are no adjacent edges in it. Similarly, there are no adjacent edges in M'. Hence all nodes in G' have degree less than or equal to two. If a node has degree two, then it is incident to one edge from M and another edge from M'. Hence all the connected components of G' are either cycles of even length or paths. Any cycle in G' has an equal number of edges from M and M'. But $|M'| > |M|$. Hence there exists a path in G' that has one more edge from M' than M. This path must start and end with edges from M' and the first and the last nodes are not incident to any edge of M. Hence this path is an augmenting path with respect to M. $\square$

If there is an augmenting path P in a graph with respect to a matching and a new matching M' is created from M by excluding the matched edges of P from M and including the unmatched edges, the size of the new matching M' is one bigger than that of M. This process is referred to as *augmenting an augmenting path*. A simple algorithm to find a maximum matching is to start with a null matching, then find an augmenting path, and augment along that path. This process can be repeated until there are no more augmenting paths.

2.2 Models of Distributed Systems

A distributed system can be modelled in various ways. In this report the network in a distributed system [14, 2] is modelled as a graph where the nodes of the graph represent the processes. If two processes can directly communicate with each other, an edge is assumed between the corresponding nodes in the graph. For the rest of this report, a graph means a network of processes and a node means a process. A process is modelled as a state machine. The *state* of a process is the value of its variables. A process changes state based on its current state and the state of one or more of its neighbours (A process reads the states of its neighbours by communicating with them. The communication process is discussed in subsection 2.2.3). The *state transition function* of a process is defined by its algorithm. The collection of the states of the processes is the *configuration of the system*. A nonempty subset of the system configurations is defined as the initial configurations. Another nonempty subset of the system configurations is defined as the final configurations. Based on the behavior of the processes and the way they communicate, distributed systems can be modelled in various ways. In this report, only one such model is used. This model and some related models and issues are described below.

2.2.1 Network topology

The network topology of a distributed system is described by the topology of the underlying graph. The network topology can be fixed, for example a ring or a grid with a fixed number of processes. But in most applications, the network is an arbitrary graph with an arbitrary number of processes. The graph may have certain properties. For example, certain applications may assume the underlying graph is a tree, is bipartite or complete. The network topology of a system can be static or it may dynamically change due to addition or deletion of nodes and communication channels. There may also be restrictions on the number of nodes in the system. The nodes of the network may have distinct identifiers. If the nodes of a distributed system do not have identifiers then the system is *anonymous*. In many systems, one particular node may be distinguished by special characteristics and act as a leader.

2.2.2 Algorithm type

Depending on its state transition function, a distributed algorithm can be *deterministic* or *randomized*. In a deterministic algorithm, the next state of a process is uniquely determined by its current state and the states of its neighbours. A randomized algorithm gives a probability distribution among a set of possible next states based on the current state of a process and its neighbours. A randomized algorithm for a problem π is a *Las Vegas algorithm* if it solves π with probability one. A randomized algorithm for a problem π is a *Monte Carlo algorithm* if it solves π with probability p, where $0 < c \leq p < 1$ and c is a constant. The randomized algorithm presented in this report is a Las Vegas algorithm.

Randomization is often employed to break the symmetry in an anonymous system.

2.2.3 Communication

In a distributed system, processes can communicate either by passing messages via a communication channel or by using a shared memory.

In a message passing model, the channel can be unidirectional or bidirectional. A message queue is assumed at the receiving end of the channel. The size of this queue can be bounded or unbounded. The communication channel may deliver the messages in the order they were sent, or the ordering may be arbitrary. A distributed system that communicate by passing messages is referred to as a message passing network.

In a shared memory model, processes communicate by reading and writing to one or more shared variables. If a shared variable is written by only one process and read by multiple processes, the model is a single-writer multi-reader model. A shared variable can be shared by all the processes or a subset of processes. In this report, all communications will be assumed to occur by single-writer multi-reader shared variables. In this model, the shared variable is owned by one process and that process can modify it, while a subset of other processes can read it. If process x can read the shared variables of process y, then it is assumed that y can read the shared variables of x and they are adjacent. This model is referred to as the *locally shared memory model*. In this model, each process has an arbitrary ordering of its neighbours. For neighbour y of x, $\text{pos}_x(y)$ denotes y's position in x's order. It is assumed that y knows its position in x's order by some other mechanism. For notational convenience, we will use y to mean $\text{pos}_x(y)$, if there is no scope for ambiguity.

Another variation of the shared memory model is the *link-register* model. In this model a pair of processes can communicate by a shared register that is accessible by those two processes only.

This shared register is called a *link-register*. A link-register can be unidirectional, meaning that only one process can write to it and the other can read; or it can be bidirectional, meaning that both processes can read from it and write to it. In the case of the unidirectional model, a pair of registers is normally associated with each communication edge, one for each direction of communication. A message passing network can simulate a link-register system with some restrictions.

2.2.4 Timing model

Depending on the timing by which the processes execute their algorithms, a distributed system can be *synchronous* or *asynchronous*.

In a synchronous message passing system, all processes send messages to all or a subset of their neighbours, receive messages from their neighbours, and then update their states based on the information received, in lock step. In a synchronous shared memory system, all processes read the shared variables, perform local computation and update shared variables, in lock step. A *synchronous round* is defined as one such step of computation by all processes.

In an asynchronous system, processes execute their algorithms at different speeds.

2.2.5 Refinements of the asynchronous locally shared memory model

In this report, we will use only the asynchronous locally shared memory model. This model and the additional issues relevant to this model are described below.

Computation step A *computation step* by a process is the amount of computation it performs in one atomic operation without being interrupted. A computation step consists of some communication operation and local computation. The *atomicity* of an asynchronous shared memory distributed system determines the granularity of a computation step. The atomicity of a system can be *read/write* or *composite*.

In a read/write atomic system, a computation step of a process consists of a single read or write to one of the shared variables, preceded or followed by some internal computation.

In a composite atomic system, in one computation step, a process can read all or any subset of the shared variables, do internal computation, and then write to one or more shared variables.

During this entire process, none of the neighbours of the process takes any step. The composite atomic model is not realistic. This model is used for the ease of designing algorithms. There are compilers that transform a composite atomic algorithm to a read/write atomic algorithm under some conditions.

A randomized algorithm can be *coarse atomic* or *fine atomic*. In a coarse atomic algorithm, in one indivisible step, a process makes a random choice and performs the next communication operation, without being interrupted. In a fine atomic algorithm, a process may pause its execution after it makes the random choice but before it makes the communication operation based on that random choice.

In an asynchronous system, processes execute their computation steps at arbitrary speeds. The amount of time taken by a computation step is assumed to be finite but unbounded.

Execution of an asynchronous algorithm A process is *enabled* in a configuration c, if it can make a state transition in c. In configuration c, let S be a subset of all enabled processes. A *move* of a system is a three-tuple, (c,S,c'), such that if all processes in S make state transitions in c, the system enters the new configuration c'.

If there are two adjacent processes in S and their computation steps involve access to the same shared variable, it is assumed that the underlying hardware serializes these read and write operations in a total order (In other words, the shared variables are assumed to have sequential memory consistency). A read from a shared variable returns the value of the most recent write to that variable in the total order.

Starting from some initial configuration c_0, a potential *execution* of a distributed algorithm is a sequence (possibly infinite) of moves

$$(c_0,S_0,c_1),(c_1,S_1,c_2),\ldots,(c_i,S_i,c_{i+1}),\ldots$$

of the system.

An *execution segment* is a contiguous subsequence of an execution. Process x takes a computation step in an execution segment $(c_l,S_l,c_{l+1}),\ldots,(c_{l+m-1},S_{l+m-1},c_{l+m}), l \geq 0, m \geq 1$, if

$$x \in \bigcup_{i=l}^{l+m-1} S_i.$$

The scheduler A *schedule of an execution* is an ordering of the moves of the execution. A *scheduler* is an objectification of the process that provides such a schedule. Informally, the scheduler chooses the sets S_i and determines the execution of the algorithm, starting from some initial configuration. A scheduler is *fair* if, in an infinite execution, a process that is enabled infinitely often is scheduled infinitely often. A scheduler is *k-fair* if it does not schedule a process more than k times between two consecutive steps of any process that is enabled continuously. When $k = 1$, the scheduler is referred to as a *round robin scheduler*. An *unfair* scheduler does not have any fairness restrictions.

In an asynchronous distributed system, processes may or may not take concurrent steps. Based on this restriction, a scheduler can be *central* or *distributed*. A central scheduler schedules exactly one process at a time to take a computation step. In the above definition of an execution, the scheduler is central if $|S_i| = 1$ for all $i \geq 0$. A distributed scheduler schedules any nonempty subset of the enabled processes at any time to take a computation step simultaneously.

In an asynchronous system there can be an external entity, for example an operating system on a single processor computer (central scheduler), that decides the order in which the processes take steps. In some systems there is no external entity for deciding the schedule. Instead, the processes independently take steps based on their machine speed; for example processes running on computers connected on an intranet (distributed scheduler).

An asynchronous distributed algorithm must work for any scheduling obeying any predefined fairness restriction. Therefore, designing and analyzing a distributed algorithm can be viewed as a game between the algorithm designer and an adversary, the scheduler.

While the read/write atomic system is more realistic than the composite atomic system, designing and analyzing algorithms for read/write atomic systems turns out to be more difficult and, in some anonymous systems, impossible. In a read/write atomic system, in one computation step, a process can access only one shared variable. Therefore, if the state transition requires the states of more than one shared variables, the process must copy the shared variables to local variables and make the state transition based on the local variables. At the time of making the state transition, the actual values of the shared variables may have changed from their local copies. This makes designing and analyzing algorithm for read/write atomic systems more difficult than composite atomic systems.

2.2.6 Fault model

A distributed system is subject to a lot of uncertainties. In particular, the processes and communication channels are prone to faults. A process or a communication channel may permanently fail by crashing. In the case of a transient fault, the faulty process or communication channel can recover from the fault. A communication channel can be lossy, meaning that it may lose a subset of the messages, or it may alter the ordering of the messages.

Fault tolerant distributed systems are built in such a way that they can withstand particular type of faults. For example, *wait free* systems can tolerate permanent or transient crash failures. In these systems, no matter how many processes or communication channels fail by crashing, the non-faulty processes terminate correctly within a finite amount of computation.

A process or communication channel may undergo more severe *Byzantine* faults that may corrupt its program. Hence, it may behave arbitrarily, even maliciously. There are algorithms that can tolerate Byzantine faults of a sufficiently constrained subset of the processes.

In some applications, reliable delivery of messages is required even when processes and communication channels fail. Fault tolerant point-to-point communication algorithms are available for these applications, for example multi-path routing algorithms.

2.2.7 Self-stabilization

The fault tolerance methods used in most distributed computing literature are piecemeal approaches to fault tolerance. A more useful approach would be a unified model of fault tolerance [16]. Recall that a distributed system starts in some initial configuration in a set of pre-defined initial configurations, and after a finite amount of computation, reaches a legitimate configuration. But a transient or permanent fault may subsequently move the system back to an illegitimate configuration. To re-converge to a legitimate configuration, the system must be brought back to some initial configuration. If instead, distributed systems are built such that the set of initial configurations is the same as the set of all configurations, then no matter what configuration the system starts from, after a finite amount of computation, the system will converge to a legitimate configuration. If a system has this property, it can withstand any type of fault, except on-going Byzantine faults. This property is referred to as self-stabilization. Since a fault can occur at any time, self-stabilizing algorithms never terminate.

Let S be a distributed system and C be the set of all configurations of S. Let P be a predicate defined over the configurations of C. Let L be the set of *legitimate configurations for predicate P*, defined as

$$L = \{c \in C | P(c)\}.$$

Then, the system S is *self-stabilizing for predicate P* [4], if it satisfies:

1. Closure: Once the system is in some configuration in L, all subsequent configurations are in L in any fault-free execution, and

2. Convergence: When started in any configuration in C, after a finite amount of computation the system reaches some configuration in L.

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.

2.2.8 Proof techniques

The correctness proof for a self-stabilizing algorithm consists of the proof of the convergence and the closure properties. In most applications, the closure property is easier to prove than convergence. There are several techniques for proving convergence of self-stabilizing algorithms, for example the progress and safety technique, and the variant function techniques.

The safety and progress technique provides a general approach for proving correctness of distributed algorithms. With this technique, it is asserted that after a bounded amount of computation, the system always has certain properties, regarded as the safety properties. For example, in the maximum matching algorithm, it is asserted that after the first round of computation, the size of the matching never decreases. The progress property of the system ensures that within a bounded amount of computation the system enters a configuration that is, in some way, closer to a legitimate configuration. For example, in the maximum matching algorithm it is asserted that the size of the matching increases within a bounded amount of computation, as long as the matching of the system is not maximum. Together, the safety and progress properties ensure that the system converges to a legitimate configuration within a bounded amount of computation.

The variant function technique is a formalization of the safety and progress technique. Let C be the set of all configurations of a distributed system, and L be the set of legitimate configurations. Let S be a totally ordered finite set with $|S| \geq 2$, and an element t, referred to as the threshold element. The variant function $f : C \to S$ is a mapping, such that

$$f(c) > t, \text{ if and only if } c \in L.$$

In this technique, to prove the safety property, it is shown that for any state transition by a process changing the system configuration from c_1 to c_2, $f(c_2) \geq f(c_1)$. For progress, it is shown that from any illegitimate configuration, within a finite amount of computation, some process makes a state transition that changes the system configuration from c_1 to c_2, such that $f(c_2) > f(c_1)$. Hence, in any fault free-execution, after a finite amount of computation, the function value corresponding to the system configuration exceeds the threshold value. At this point the system is in a legitimate configuration and remains in a legitimate configuration. Finding such a variant function is typically not trivial and requires a lot of intuition.

2.2.9 Fair composition

Fair composition is a technique introduced by S. Dolev *et al.*[5] to design and analyze self-stabilizing solutions for complex problems.

To describe the concept of fair composition, generalize the definition of self-stabilizing systems to stabilizing systems [1]. Let C be the set of configurations for the system S. Let P and Q be two predicates defined over the configurations of S. Let L_P be the set of legitimate configurations for predicate P, defined as

$$L_P = \{c \in C | P(c)\}.$$

Let L_Q be the set of *legitimate configurations for predicate Q, given predicate P*, defined as

$$L_Q = \{c \in L_P | Q(c)\}.$$

Then algorithm A_Q, for system S, is *stabilizing for predicate Q, given predicate P* if,

1. Closure: Once S is in a configuration in L_Q, all subsequent configurations of S are in L_Q, in any fault free execution of A_Q.

2. Convergence: Starting from any configuration in L_P, S converges to a configuration in L_Q within a finite amount of computation, in any fault-free execution of A_Q.

Let algorithm A_P, for the system S, be self-stabilizing for predicate P. Also, assume that algorithm A_P does not use the variables of A_Q, and algorithm A_Q does not modify variables of A_P. Then it can be shown [4, pp. 22-24] that if an algorithm A is built by composing A_P and A_Q, in which the processes of system S run the algorithms A_P and A_Q alternately (take one computation step of each algorithm alternately), then the composed algorithm A is self-stabilizing for predicate Q. The composed algorithm is called a fair composition of algorithms A_P and A_Q.

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

2.2.10 Complexity measures

There are two kinds of complexity measures for distributed algorithms, *time complexity* and *space complexity*.

Time complexity measures how long a distributed algorithm takes to solve a problem. For deterministic algorithms, the average-case and the worst-case behavior of an algorithm are measured. In the case of randomized algorithms, the time complexity is the expected amount of time taken to solve a problem. A self-stabilizing algorithm does not terminate even after it converges to a legitimate configuration. Instead, a self-stabilizing algorithm iterates through its algorithm in a do-forever loop. Hence, in the case of self-stabilizing algorithms, the most important parameter is the amount of time taken by the algorithm to converge to a legitimate configuration. Therefore, the time complexity of a self-stabilizing algorithm refers to the time taken by the algorithm to converge to a legitimate configuration.

In an asynchronous distributed system, there is no notion of real time since no upper bound is assumed for the time taken by a computation step. However, to evaluate and compare algorithms, some notion of efficiency is necessary. One way of measuring efficiency of an algorithm is by counting the number of asynchronous rounds it takes to converge to a legitimate configuration. Recall that an execution E is a sequence of moves by the system. The asynchronous round 0 of computation of an execution E is the empty execution prefix of E. The asynchronous round $i(\geq 1)$ of computation is defined inductively as follows. Let $E = E'E''$, such that E' is the shortest prefix of E containing the computation up to round $i-1$. Then asynchronous round i of computation is the shortest prefix of E'' containing at least one computation step by every enabled process. Note that in any one round, some processes may take multiple computation steps. The round complexity of an execution is the number of asynchronous rounds in it.

In the case of a composite atomic algorithm, one iteration of the algorithm consists of a single computation step. In the case of a read/write atomic algorithm, one iteration of the algorithm may consist of several computation steps. Generally, in one iteration of the algorithm, a process reads the states of its neighbours, performs local computations and updates its own state. Hence one iteration by a process may consist of as many as O(Δ) reads and a constant number of writes, where Δ is the maximum degree of the system. However, in some cases, it is much easier to count the number of complete iterations made by a process. In these cases, the complexity of an algorithm is measured in terms of asynchronous cycles. The asynchronous cycle 0 of computation of an execution E is the empty execution prefix of E. The asynchronous cycle $i(\geq 1)$ of computation is defined inductively as follows. Let $E = E'E''$, such that E' is the shortest prefix of E containing the computation up to cycle $i-1$. Then asynchronous cycle i of computation is the shortest prefix of E'' containing at least one complete iteration by every process. The cycle complexity of an execution is the number of asynchronous cycles in it.

For a composite atomic algorithm, the round complexity and cycle complexity are the same. For read/write atomic algorithms, the round complexity is no bigger than O(Δ) times the cycle complexity. For networks where Δ is a constant, both complexities agree asymptotically.

The round complexity of an algorithm gives an estimate of computation time when the scheduler schedules processes evenly. But in a round, some process may take multiple steps, while another process takes just one step. Therefore, when the scheduler schedules some set processes more often than other processes, the round complexity does not reflect the actual computation time. In this case the computation time is better estimated by measuring the step complexity. The *step complexity* of an execution is the total number of state transitions made by all processes in that execution.

For a deterministic algorithm, the worst-case complexity of the algorithm is the maximum number of rounds, cycles or state transitions taken by the algorithm to converge to a legitimate configuration over all possible executions, starting from any initial configuration. Defining the expected round complexity of a randomized algorithm is a little difficult, since the number of rounds taken by the algorithm to converge to a legitimate configuration depends both on the scheduling and the random choices. Starting from some initial configuration, given a sequence of random choices, there is a worst case execution taking the maximum number of rounds before the system converges to a legitimate configuration. The expected round complexity of the algorithm from that initial configuration is the weighted average of the number of rounds taken by the worst case execution over all possible sequences of random choices. The expected round complexity of the algorithm from any initial configuration is the maximum of the expected round complexity over all possible initial configurations. The expected cycle complexity of an algorithm is defined similarly. The complexity of an algorithm is specified as a function of the parameters of the network, such as the number of processes, the number of edges and the diameter of the network.

The *space complexity* of a shared memory algorithm measures the total number of memory bits (local and shared) used by all processes.

Chapter 3

Related Work

Finding a maximal or a maximum matching is a well studied problem in graph theory, for which several efficient centralized algorithms [6, 11] 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.

3.1 Previous Maximal Matching Algorithms

Hsu and Huang [12] present a self-stabilizing distributed algorithm for finding a maximal matching in a general network. The model of computation is locally shared memory under composite atomicity and a centralized scheduler. Each process has a shared variable, referred to as a pointer. The pointer may be null or may point to a neighbouring process. If two neighbouring processes point at each other, the processes are matched along that edge. The algorithm works in anonymous networks, although it implicitly assumes some kind of labelling that allows a process to determine where its neighbours are pointing. It does not require any knowledge of the network size. They use an elegant variant function technique to prove convergence in $O(n^3)$ state transitions, where n is the number of nodes in the system. However the upper bound is not tight. Tel [20] uses a slightly different variant function to prove the upper bound to be $O(n^2)$. Hedetniemi, Jacobs and Srimani [9] use a different proof technique, which counts the number of possible moves along any edge, to prove the bound to be $\Theta(2m+n)$, where m is the number of edges in the system.

The algorithm by Hsu and Huang does not work under a distributed scheduler. In fact an easy symmetry breaking argument shows that it is impossible to solve the problem deterministically in an anonymous general graph under a distributed scheduler. Gradinariu and Johnen [7] overcome this problem by randomization. They first present an algorithm to assign locally distinct labels within distance two using randomization. The model of computation of this algorithm is locally shared memory under composite atomicity and a distributed scheduler. They then use this to extend the Hsu-Huang maximal matching algorithm to work under a distributed scheduler assuming locally distinct labels within distance two. The model of computation of this algorithm is also locally shared memory under composite atomicity and a distributed scheduler. A fair composition of these two algorithms solves the maximal matching problem in an anonymous general network.

The open problem remaining after this work is to solve maximal matching in an arbitrary anonymous network under read/write atomicity.

3.2 Previous Maximum Matching Algorithms

Karaata and Saleh [13] present a self-stabilizing distributed algorithm for finding a maximum matching in a tree. The algorithm works in two phases. In the first phase, the undirected tree is converted to one or two directed trees rooted at one or two distinguished processes, referred to as *centers*. The second phase uses a greedy approach. First, each leaf node is matched with its neighbour and the matched nodes become disabled. Therefore, the matched nodes and the incident leaf nodes can be assumed to be removed from the system, resulting a smaller system. This process is repeated until there are no more edges. It can be shown that at this time, there are no more augmenting paths. Hence the system converges to a maximum matching. The correctness proof of the algorithm assumes composite atomicity and a centralized scheduler, but identifiers are not needed. The algorithm uses $O(n^4)$ state changes in the worst case.

3.3 Previous 2-coloring Algorithms

The maximum matching algorithm presented in this report works in a bipartite graph under the assumption that the processes know their bipartition. A self-stabilizing 2-coloring algorithm can be used by the processes to find their bipartition. Then a fair composition of these two algorithms may be used to solve the maximum matching problem in an arbitrary bipartite graph.

Sur and Srimani [19] present a self-stabilizing distributed algorithm for 2-coloring a bipartite graph. This algorithm needs a distinguished process (a leader). First, processes determine their distance from the leader in a breadth first manner. Processes at an even distance from the leader colour themselves black and processes at an odd distance from the leader colour themselves white. The proof of correctness assumes a central scheduler and composite atomicity. However, it can be easily shown that the algorithm is correct even under a distributed scheduler and read/write atomicity. The correctness proof shows that the algorithm converges to a 2-coloring within a finite amount of computation, but does not provide any upper bound on the amount of computation.

Shukla, Rosenkrantz and Ravi [17, 18] prove that there is no deterministic self-stabilizing algorithm for 2-coloring general anonymous bipartite graphs under a distributed scheduler. The impossibility result follows directly from a simple symmetry breaking argument. They then present self-stabilizing 2-coloring algorithms for restricted classes of graphs: paths of odd and even length, oriented rings, and complete bipartite graphs of odd degree. The papers are extended abstracts that do not provide detailed proofs of correctness or complexity analysis. However, in all cases, the correctness proof and analysis are simple and straightforward.

Chapter 4

Maximal Matching

4.1 Introduction

Our goal is to find a self-stabilizing algorithm that solves the maximal matching problem in a general anonymous network, under read/write atomicity and a distributed scheduler. The following lemma shows that this cannot be done deterministically. The proof employs an easy symmetry breaking argument, widely used in distributed computing literature.

Lemma 4.1.1. *Deterministic distributed maximal matching under read/write atomicity is impossible in an arbitrary anonymous network.*

Proof. Consider a triangular network: three processes, every pair connected. Any configuration in this system, satisfying a predicate for maximal matching, must have two of the processes matched and the third process unmatched. Hence, in any legitimate configuration in this system satisfying a predicate for maximal matching, the processes' states cannot be symmetric. If all the processes start as unmatched, some processes must write to one or more of their variables. Suppose the scheduler runs the processes one at a time, stopping each just before it writes to one of its variables. Then it lets each process write. Because the processes start with identical states and their neighborhoods are identical, each process writes the same value. Hence after this write operation, all the processes are again in identical states. The scheduler can keep doing this forever and keep the states of all processes identical after each such cycle. Hence the symmetry is never broken. ☐

Note that a similar argument can be applied to a message passing system. Further note that a self-stabilizing system can start in the same configuration as an ordinary distributed system, so the same argument can be applied, implying that there is no self-stabilizing algorithm for distributed maximal matching under read/write atomicity in an arbitrary anonymous network.

This impossibility can be circumvented with randomization. We first present a randomized algorithm that assigns locally distinct labels within distance two [8] [1]. We then present an algorithm that uses these labels to solve the maximal matching problem. A fair composition of these two algorithms solves the maximal matching problem in anonymous general networks under read/write atomicity and a distributed scheduler.

4.2 Randomly Generating Locally Distinct Labels

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

4.2.1 Model

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

4.2.2 Data structures

In the following descriptions, a process's name is used for notational purposes only. Each process x maintains a shared label called lid and a shared variable δ_x, which are positive integers. Process x has a shared array H_x of sets of labels, where each set is indexed by $\text{pos}_x(y)$ for each process y in $N(x)$. Since there is no scope of ambiguity, $\text{pos}_x(y)$ is denoted as y in the following sections. Let $N(x) = \{y_1, \ldots, y_m\}$. Then

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

[1]This paper provides interesting results about the number of labels necessary for locally distinct labelling of a graph within distance two, for a variety of different types of graphs.

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

Process x has local variables Δ_x holding a positive integer, A holding a set of labels, and M holding a multiset of labels. Let $\mathrm{mult}(S,x)$ denote the multiplicity of x in multiset S. Multisets $S+x$ and $S-x$ are identical to S except for the multiplicity of x. The multiplicity of x in $S+x$ is $\mathrm{mult}(S,x)+1$, and the multiplicity of x in $S-x$ is $\max\{0,\mathrm{mult}(S,x)-1\}$.

When a 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 $T \leftarrow S$.

4.2.3 Informal description of the algorithm

Process x uses the shared variable δ_x to inform its neighbours about the size of its neighbourhood. A process determines the maximum neighbourhood size in its closed neighbourhood from the shared δ variables and stores that in the local variable Δ_x. Then it uses this value to determine the range of the labels, which is $[1, f(\Delta_x)]$. The function f will be defined later.

Each process collects the lid's of the processes in its closed neighbourhood in the multiset M. Then it builds each set of the array H_x for each of its neighbours after removing that neighbour's lid from the multiset M. The purpose of these sets is to inform a process about the lid's of the processes within a distance two.

Process x collects, for each $y \in N(x)$, the set $H_y[x]$, and stores the values in the set A. It then checks if its own lid appears in A. If it does, the process randomly chooses a new label.

4.2.4 Algorithm for process x

```
do forever
    δ_x ← |N(x)|
    A ← ∅
    M ← ∅
    Δ_x ← δ_x
    for all y ∈ N(x)
        A ← A ∪ H_y[x]
        M ← P + y.lid
        Δ_x ← max{Δ_x, δ_y}
    if (there exists z ∈ A such that z = x.lid) then
            randomly choose x.lid uniformly from 1 to f(Δ_x) that is not in A
    P ← M + x.lid
    for all y ∈ N(x)
        H_x[y] ← M − y.lid
```

Figure 4.1: `Label generation algorithm`

4.2.5 Analysis and correctness

For process x, define

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

The set $L(x)$ contains: (i) the actual labels of all neighbours of x within distance two; (ii) the labels in the set for x in the array of each neighbour of x; (iii) the labels in x's local set A. Define x to be *secure* if $x.lid \notin L(x)$; otherwise x is *insecure*.

For any configuration c, define predicate $P(c)$ to be: all processes are secure in c. Note that in any configuration satisfying P, processes have locally distinct labels within distance two. We will prove that the `Label Generation Algorithm` is self-stabilizing for predicate P.

Observation 4.2.1. *After the first cycle, the shared variable δ_x of process x always holds the size of its neighbourhood. Starting from the second cycle, in every cycle, process x correctly determines the maximum degree in its closed neighbourhood and stores it in Δ_x. For the rest of the proof, Δ_x represents the maximum degree in x's closed neighbourhood. Hence $\delta_x \leq \Delta_x$.*

Note that the size of $L(x)$ is less than $3\Delta_x^2$.

Claim 4.2.2. *If an insecure process x chooses a label and is still insecure, it causes at most δ_x secure processes to become insecure. This happens with probability less than $\frac{3\Delta_x^2}{f(\Delta_x)}$.*

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

There are less than $3\Delta_x^2$ elements in the set $L(x)$ for process x. A process x chooses a new label uniformly from the range $[1, f(\Delta_x)]$. Hence the probability that process x chooses a label in $L(x)$ and stays insecure is less than $\frac{3\Delta_x^2}{f(\Delta_x)}$. ☐

Claim 4.2.3. *An insecure process cannot remain insecure continuously for more than two cycles without choosing a new label.*

Proof. If process x is insecure because x.lid $\in A$, then process x either updates A and becomes secure or chooses a new label in the current cycle. If it is insecure because x.lid $\in H_y[x]$ for some process $y \in N(x)$, then either y updates $H_y[x]$ and x becomes secure or x.lid appears in set A in the next cycle and x chooses a new label in the same cycle. If it is insecure because x.lid $= y$.lid for some process $y \in N_2(x)$, then either x becomes secure because y chooses a new label different from x.lid, or x.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 cycles. ☐

Theorem 4.2.4. *If $f(\Delta) \geq 6\Delta^2(\Delta + 1)$, then starting from an arbitrary configuration the system converges to a legitimate configuration within an expected $2\log_2 n + 5$ cycles, where n is the number of processes in the system.*

Proof. The proof is motivated by the standard randomized attrition technique [10].

Define a super-cycle as two consecutive cycles of computation. Define a random variable X_i as the number of insecure processes in the system at the start of super-cycle $i+1$. If at the beginning of super-cycle $i+1$ there are l insecure processes, say processes $1,\ldots,l$, in the system, then by Claim 4.2.3, each one of these process either becomes secure or chooses a new label within one super-cycle. When an insecure process $j, 1 \leq j \leq l$ chooses, it stays insecure with probability less than $\frac{3\Delta_j^2}{f(\Delta_j)}$. Hence the expected number of processes among those l insecure processes that can stay insecure is less than

$$\sum_{i=1}^{l} \frac{3\Delta_j^2}{f(\Delta_j)}.$$

Since each process $j, 1 \leq j \leq l$ can make at most Δ_j secure processes insecure (Claim 4.2.2),

$$\begin{aligned}
E(X_{i+1}|X_i = l) &< \sum_{j=1}^{l} \frac{3\Delta_j^2}{f(\Delta_j)} + \sum_{i=1}^{l} \Delta_j \frac{3\Delta_j^2}{f(\Delta_j)} \\
&< \sum_{j=1}^{l} \frac{3\Delta_j^2(\Delta_j+1)}{f(\Delta_j)} \\
&< \sum_{j=1}^{l} \frac{3\Delta_j^2(\Delta_j+1)}{6\Delta_j^2(\Delta_j+1)} \\
&\quad \text{Since } f(\Delta) \geq 6\Delta^2(\Delta+1) \\
&= \sum_{j=1}^{l} 1/2 \\
&= l/2
\end{aligned}$$

$$\begin{aligned}
E(X_{i+1}) &= \sum_{l \geq 0} E(X_{i+1}|X_i = l) Pr[X_i = l] \\
&< \sum_{l \geq 0} \frac{l}{2} Pr[X_i = l] \\
&= \frac{1}{2} E(X_i)
\end{aligned}$$

Thus if the system starts with m $(\leq n)$ insecure processes, then after $\log_2 m$ super-cycles, the expected number of insecure processes is reduced to at most one. Let $E(i,j), 0 \leq j \leq i \leq n$, denote the expected number of cycles for the number of insecure processes to go down from i to at most j. Then $E(m,1) = \log_2 m$, and if there is only one insecure process, say x,

$$
\begin{aligned}
E(1,0) &\leq 1\cdot\left(1-\frac{3\Delta_x^2}{6\Delta_x^2(\Delta_x+1)}\right)+\frac{3\Delta_x^2}{6\Delta_x^2(\Delta_x+1)}E(\Delta_x,0)\\
E(1,0) &\leq \left(1-\frac{1}{2\Delta_x+2}\right)+\frac{1}{2\Delta_x+2}\left(E(\Delta_x,1)+E(1,0)\right)\\
E(1,0) &\leq \left(1-\frac{1}{2\Delta_x+2}\right)+\frac{1}{2\Delta_x+2}\left(\log_2\Delta_x+E(1,0)\right)\\
(2\Delta_x+1)E(1,0) &\leq 2\Delta_x+1+\log_2\Delta_x\\
E(1,0) &\leq 1+\frac{\log_2\Delta_x}{2\Delta_x+1}\\
&< 2.
\end{aligned}
$$

Thus, if there is only one insecure process, then within an expected two super-cycles all processes become secure. Hence the system converges to a legitimate configuration within an expected $2\log_2 n+5$ cycles. □

Theorem 4.2.5. *If $f(\Delta)\geq 6\Delta^2(\Delta+1)$, then the expected number of random choices by all processes until the system converges to a legitimate configuration is at most $2n$.*

Proof. Initially the system may start with some insecure processes. By Claim 4.2.3, an insecure process, say x, either becomes secure or makes a random choice. If it makes a random choice, it either becomes secure or stays insecure and makes at most Δ_x additional secure processes insecure. The probability p_x that x becomes secure is greater than $1-\frac{3\Delta_x^2}{f(\Delta_x)}\geq 1-\frac{3\Delta_x^2}{6\Delta_x^2(\Delta_x+1)}$. Probability q_x that x stays insecure is less than $\frac{3\Delta_x^2}{f(\Delta_x)}\leq\frac{3\Delta_x^2}{6\Delta_x^2(\Delta_x+1)}$. We want an upper bound on the expected number of random choices before all processes become secure. This expectation is maximized by assuming that initially all processes are insecure, and when a process makes a random choice and stays insecure, it makes any Δ_x secures processes insecure.

Let $E(X_i)$ denote the expected number of secure processes before the i'th random choice.

$$
\begin{aligned}
E(X_{i+1}|X_i=l) &\geq (l+1)p_x+(l-\Delta_x)q_x\\
&= (p_x+q_x)l+p_x-q_x\Delta_x\\
&\geq l+1-\frac{1}{2(\Delta_x+1)}-\frac{\Delta_x}{2(\Delta_x+1)}\\
&= l+1/2
\end{aligned}
$$

This expectation is independent of l and x. Hence, starting from any initial condition, within expected $2n$ random choices, all processes become secure. $\square$

Convergence: Both Theorem 4.2.5 and Theorem 4.2.4 establishes that, starting from an arbitrary configuration, the system converges to a legitimate configuration within a finite amount of computation.

Closure: Note that if all processes are secure, the condition for the choice operation is false for all processes, and no process chooses a new label. Hence once the system reaches a legitimate configuration, processes' labels do not change and all processes remain secure in any subsequent configuration.

Complexity: Theorem 4.2.4 establishes that the expected number of cycles before the system converges to a legitimate configuration is at most $2\log_2 n + 5$.

For determining the step complexity of the algorithm, note that shared variable H is an array of the shared variables lid. The change in the variable lid by a process can cause state change of all its neighbours due the changes in their H variables. A process can have at most Δ neighbours. Hence, the total number of state transitions by all processes of the system is in the order of Δ times the total number of changes to the shared variables lid. By Theorem 4.2.5, the expected number of changes to the shared variables' lid is less than $2n$. Hence, the step complexity of the algorithm is in $O(n\Delta)$.

4.3 Maximal Matching

This section presents a stabilizing algorithm for finding a maximal matching in an arbitrary network, given locally distinct labels within distance two.

4.3.1 Model

The network is a general graph. The model of computation is locally shared memory under read/write atomicity with a distributed scheduler. Processes do not need any global information about the network topology. The network has locally distinct labels within distance two. Labels are chosen from the set of positive integers $[1, R]$, and are stored in the shared variables lid.

4.3.2 Informal description of the algorithm

The algorithm implements, in a distributed manner, the following graph theoretic scheme for finding a maximal matching.

Suppose that the nodes of a graph have locally distinct labels within distance two, and that the labels are chosen from a totally ordered set. Then the graph has local minima within distance 2 (defined below). Match a local minimum within distance 2 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.

4.3.3 Algorithm for process x

Process x has a shared variable called pref (short form for preference) that takes its value from the set of non-negative intergers $[0, R]$. It also has a local variable S_x, which is a set of labels.

```
do forever
    S_x ← ∅
    for all y ∈ N(x)
        if y.pref = 0 or y.pref ≥ x.lid
            S_x ← S_x ∪ {y.lid}
    if S_x = ∅
        x.pref ← 0
    else
        x.pref ← min{S_x}
```

Figure 4.2: Maximal matching algorithm

4.3.4 Analysis and correctness

A process x is a *local minimum within distance 2* if $x.\text{lid} < y.\text{lid}$ for all $y \in N_2(x)$.

To prove the stabilizing property of this algorithm, instead of giving a predicate for legitimate configurations, we explicitly define the set of legitimate configurations. The set contains only one configuration, described below.

Let x be a local minimum within distance 2 and $y \in N(x)$ the neighbour of x holding the minimum label among all processes in $N(x)$. Set $x.\text{pref} = y.\text{lid}$ and $y.\text{pref} = x.\text{lid}$. Remove processes x and y and the edges incident to them from the system. Repeat the above process until there are no more edges. Set the preference variables of the remaining processes (if any) to 0. The resulting configuration is the only legitimate configuration of the system, denoted c_M. Note that configuration c_M is unique, given a network with locally distinct labels within distance 2. We will prove that, given locally distinct labels within distance two, the algorithm stabilizes to c_M.

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

Lemma 4.3.1. *The set M is a matching.*

Proof. A process 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. □

If for processes x and y, $xy \in M$, then x and y are *matched* with each other.

Note that in c_M, M is a maximal matching.

Theorem 4.3.2. *Starting from an arbitrary configuration, the system converges to c_M within $2r+2$ cycles, where $r = min\{R, n/2\}$, and n is the number of processes in the system.*

Proof. After one cycle, $x.\text{pref} = 0$ or $x.\text{pref} \in N(x)$ for each process x. Suppose x is a local minimum within distance 2, and let y be the process with minimum label in $N(x)$. Since x is a local minimum within distance 2, $x.\text{lid} < z.\text{lid}$ 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 cycle and this never changes. Furthermore, $y.\text{pref} = x.\text{lid}$ after the second cycle (since x is a local minimum within distance 2) and this never changes. After the second cycle, let $z \in N(x) - \{y\}$. Then $z.\text{lid} > y.\text{lid} = x.\text{pref}$, so $x.\text{lid} \notin S_z$, and $z.\text{pref}$ is never 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 $u.\text{pref}$ is never set to $y.\text{lid}$.

Therefore, after the first two cycles, a local minimum within distance 2, x, is matched with its neighbour with minimum label, y, and these matches never change. Also, no neighbour of x, other than y ever tries to prefer x. Similarly, no neighbour of y, other than x ever tries to prefer y. The algorithm now proceeds as if the local minima within distance 2 and their neighbours holding the minimum label along with the edges incident upon them have been removed from the system. In the resulting system, the labels of the processes are in the set $[2,R]$ and the number of remaining processes is at most $n-2$. A similar argument can be applied to show that any process with label 2 is either isolated after the third cycle, or gets matched and is removed from the system after cycle 5 and the number of remaining processes is at most $n-4$. If $R \leq n/2$, it follows by induction on R that after $2R+1$ cycles, any process is either matched and removed from the system, or is isolated. If $n/2 \leq R$, it follows by induction on n that after $n+1$ cycles, any process is either matched and removed from the system, or is isolated. After that, the preference variable of each isolated process is set to 0 in one cycle. Hence the system converges to c_M within $2r+2$ cycles. □

The closure property can be established by noting that in c_M, no process can change its preference.

4.3.5 A worst case execution

Let the network be a path of processes labelled $1,\ldots,n$, where the process labelled i is adjacent to the process labelled $i+1$ for $1 \leq i \leq n-1$. Let all processes, except the process labelled 2, start with preference value zero. The process labelled 2, starts with preference value 3. Then the following schedule takes $n/2$ cycles to stabilize to a maximal matching. In this schedule, when the scheduler schedules a process, it performs a complete iteration of its algorithm. For example, when process labelled n is scheduled for the first time, it reads the states of all its neighbours (there is only one) performs local computation and then sets the pref variable to $n-1$. The scheduler then schedules the process labelled $n-1$.

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

At this point the preference of process labelled i is set to $i-1$ for $3 \leq i \leq n$. Preference value of the process labelled j is set to $j+1$ for $1 \leq j \leq 2$.

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

This is the end of the first cycle and processes 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 processes labelled $n, n-1, \ldots, 4, 3$.

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

At this point the preference of process labelled i is set to $i-1$ for $5 \leq i \leq n$. Preference value of the process labelled j is set to $j+1$ for $3 \leq j \leq 4$. The reduced system is now in a similar configuration as the original system after run one. This can be repeated $n/2$ times to give an $n/2$ cycle execution.

4.4 Fair Composition

The `Label Generation Algorithm` is self-stabilizing, and in any legitimate configuration, the system has locally distinct labels within distance two. The `Maximal Matching Algorithm` stabilizes to a maximal matching given locally distinct labels within distance two. The `Label Generation Algorithm` does not use the variables of the `Maximal Matching Algorithm`, and the `Maximal Matching Algorithm` reads the variables of the `Label Generation Algorithm`, but does not modify them. Hence a fair composition of these two algorithms is self-stabilizing for maximal matching in an anonymous general network under a distributed scheduler and read/write atomicity.

Complexity of the composed algorithm: The locally distinct labels are chosen from the set of positive integers $[1, 6\Delta^2(\Delta+1)]$. In networks where Δ is a constant, by Theorem 4.3.2, the maximal matching algorithm stabilizes within a constant number of cycles. However, the label generation algorithm itself takes O($\log n$) expected number of cycles. Thus, the composed algorithm has O($\log n$) expected cycle complexity. For graphs with $\Delta \in$ O(n), the following example shows that the composed algorithm takes O(n) cycles to stabilize.

Consider a network that consists of a path of processes labelled $2, \ldots, n$, where the process labelled i is adjacent to the process labelled $i+1$ for $2 \leq i \leq n-1$, and another process labelled 1, which is adjacent to all other processes. The system starts as if the label generation algorithm has stabilized to the above labelling, and processes labelled 1 and 2 are matched with each other. Let all other processes start with preference value zero. Then the execution shown in Subsection 4.3.5 takes O(n) cycles for the maximal matching algorithm to stabilize.

The worst-case example shown above is possible in a self-stabilizing system, but is not likely. Generally, the system starts in a configuration where most of the processes do not have any label and labels are distributed by the `Label generation algorithm`. In this case, the probability of a long chain of processes having increasing labels without an intervening local minimum is low.

Chapter 5

Maximum Matching

This chapter presents a self-stabilizing algorithm for finding a maximum matching in a bipartite graph. First we present self-stabilizing algorithm for 2-colouring a bipartite network, where the processes have distinct identifiers. Then we present a self-stabilizing maximum matching algorithm in an anonymous bipartite network, where each process knows its bipartition. A fair composition of these two algorithms solves the bipartite maximum matching problem.

5.1 A 2-colouring Algorithm

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

5.1.1 Model

The model of computation is locally shared memory, under read/write atomicity and a distributed scheduler. All processes have distinct id's. Every process has an upper bound N for the number of processes in the system. The network is an arbitrary bipartite graph.

5.1.2 Informal description of the algorithm

The basic idea is to propagate the colour and id of the process with minimum id. To do so, every process maintains, along with its colour, an origin field containing the id of the source of the colour (i.e. the id of the process that was used to choose a colour). Because this is a self-stabilizing algorithm, the system may 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 process from its origin. When the process with minimum id sets its colour to U and sets its origin to itself, it also sets its distance to 0. When a process 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 removed and the system converges to a 2-colouring within $D+1$ cycles, where D is the diameter of the graph.

5.1.3 Definitions and data structures

The state of a process is a 4-tuple (id, colour, origin, distance); id is the read-only identifier of a process and can be any positive integer; colour is either U or V; orig (short form of origin) of a process is null (denoted 0), or an id of a process; dist (short form of distance) is the distance of the process 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 process x, let $C(x) = \{y \in N(x) \mid y.\text{orig} \neq 0 \text{ and } y.\text{orig} < x.\text{id} \text{ and } y.\text{dist} \leq N-2\}$. Note that to construct $C(x)$, process x needs to read the states of all its neighbours. 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).

5.1.4 Algorithm for process x

```
do forever
  construct C(x)
  construct m(x)
  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
```

Figure 5.1: `2-colouring Algorithm`

5.1.5 Analysis and correctness:

Let π be the process with the minimum identifier. The predicate P is defined over the configurations of the system as follows. For any configuration c, $P(c)$ is true if and only if for any process x, $x.\text{dist} = d(\pi, x)$, $x.\text{colour} = U$ if $d(\pi, x)$ is even, $x.\text{colour} = V$ if $d(\pi, x)$ is odd, and $x.origin = \pi.\text{id}$. Note that there is only one configuration satisfying P and in that configuration, the network is properly 2-coloured. We will prove that the `2-colouring Algorithm` is self-stabilizing for predicate P.

An origin of a process x is *real* if $x.\text{orig} = 0$ or there exists a process y, such that $x.\text{orig} = y.\text{id}$. If the origin of a process is not real, then it is *spurious*.

Lemma 5.1.1. *After N cycles, there is no spurious origin in the system.*

Proof. Note that in every cycle, a process sets all of its variables. The proof proceeds by induction on the value of the distance field.

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

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

Lemma 5.1.2. *If there there is no process with a spurious origin, then the system converges to the legitimate configuration within $D+1$ cycles.*

Proof. Suppose that the system has no spurious origin and let π be the process with the minimum id in the system. After the first cycle, $m(\pi)$ is always 0. Hence $\pi.color = U$, π.dist $= 0$ and π.orig $= \pi$.id, and this never changes.

A process x is *properly coloured* with respect to process π if x.dist $= d(\pi, x), x$.orig $= \pi$.id and

$$x.\text{colour} = \begin{cases} U & \text{if } d(\pi, x) \text{ is even} \\ V & \text{if } d(\pi, x) \text{ is odd} \end{cases}$$

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

Theorem 5.1.3. *Starting from an illegitimate colouring, the system converges to the legitimate configuration within $N+D+1$ cycles.*

Proof. The proof follows immediately from Lemma 5.1.1 and 5.1.2. □

The above theorem proves the convergence property of the `2-colouring Algorithm`. The closure property is established from the fact that no process can change the value of its variables, once the system converges to the legitimate configuration.

5.2 Maximum Matching

5.2.1 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 process knows its bipartition and some upper bound N on the number of processes in the network.

5.2.2 Informal description of the algorithm

The protocol is motivated by the well-known sequential maximum matching algorithm for bipartite graphs [15, pp. 218-224]. In this algorithm, 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 maximum. To implement this idea in a distributed asynchronous network, the processes themselves must identify non-intersecting augmenting paths and then augment along them.

The state of a process, 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 process is bounded and a process can be modelled as a finite state machine. For process x, each of the preference and the choice fields is either null, or is a pointer to one of x's neighbours. Informally, the preference field is intended to record with which neighbour a process is currently matched, if any, and the choice field records a possible new match arising from an augmentation process. The augment-status is either silent, requesting, acknowledging or changing, and conveys how a process 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 processes from identify an alternating cycle instead of an alternating path when searching for an augmenting path.

In each composite step, a process reads all its neighbours' states and then performs a sequence of four internal steps. Initially, the state of a process 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 process 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 process 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 eventually augments the matching if an augmenting path exists.

For the description of Step 4 it is helpful to imagine that a process "sends" information to its neighbours and "propagates" information through the network, although this is achieved by a process's state changes as observed by its neighbours. The protocol is initiated by any unmatched U process. Such a process (a source) sends a request to each of its neighbours (necessarily V processes). Each such request propagates along alternating paths that may branch at U processes in the network and forms a tree structure (a requesting tree) rooted at the source. If a requesting branch encounters an unmatched V process (a sink), then the path through this requesting tree from the sink back to the source is an augmenting path. Sinks try to reserve an augmenting path. To do so, each sink initiates an acknowledgment that is sent toward the source along the alternating path in the requesting tree, reserving the processes 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 have only local information about the tree structure. Specifically, each process knows its parent, and as a potential augmenting path is being constructed, each U process knows its chosen child. (Parents and children are defined by the state of a process 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

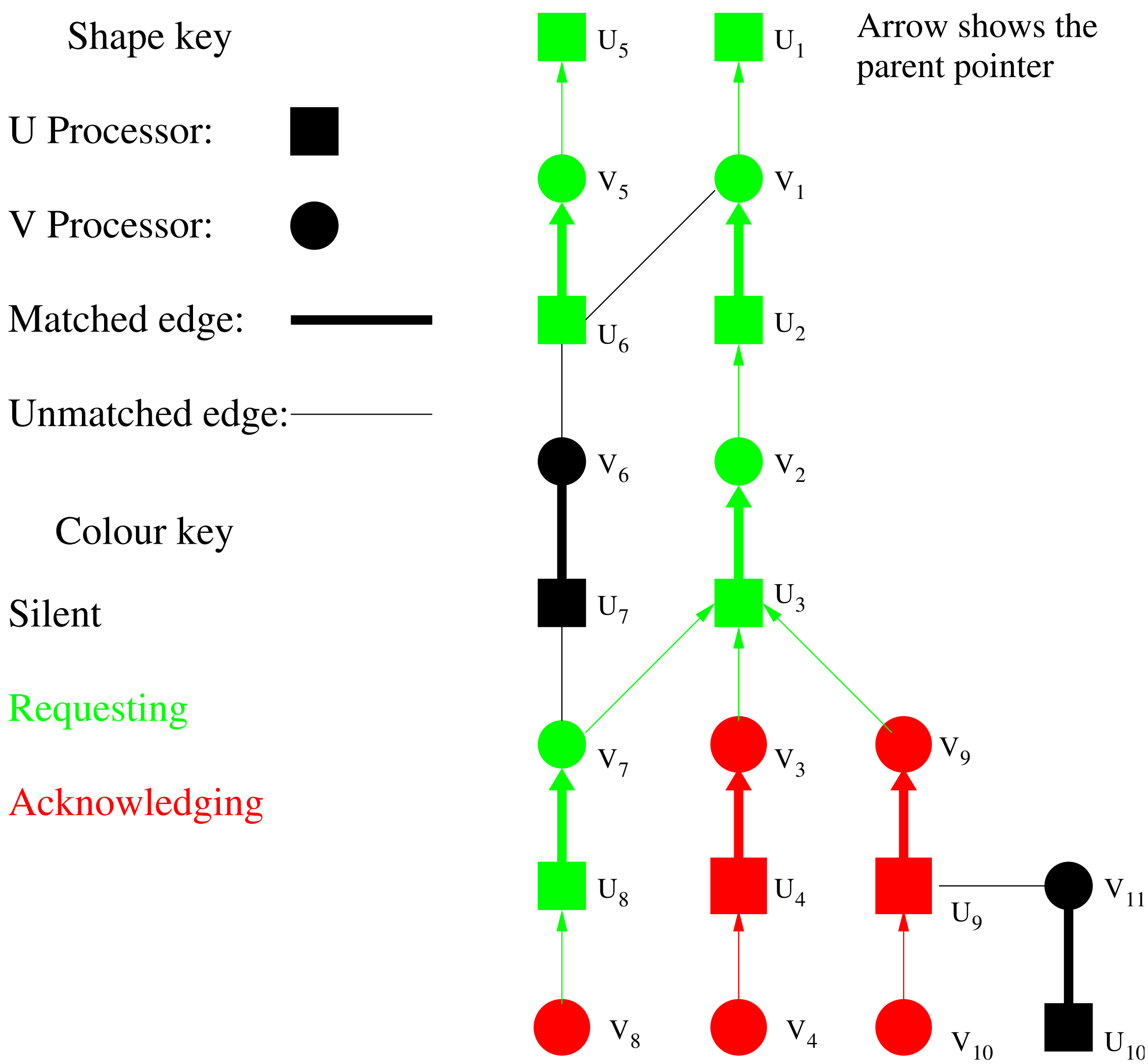

Figure 5.2: A requesting tree

pruned piece to its descendants. It is also possible that a reset process 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.

For example, in Figure 5.2, if process U_3 accepts the acknowledgment of V_3 and chooses process V_3 as its child, then the requests of processes V_7 and U_8 become spurious and the processes V_8, V_9, U_9 and V_{10} become orphans. But this guarantees the existence of the partially acknowledged augmentation path $U_1, V_1, U_2, V_2, U_3, V_3, U_4, V_4$.

In the pruned subtree, rooted at V_7, only V_7 can detect that it has been pruned. So V_7 will reset to silent. This reset may propagate to V_8; or, depending on the scheduler, a request may arrive at U_7 from the source U_5, and V_7 may attach itself to this new requesting tree. Note that for this to happen, the distance of V_7 from the old source U_1, must be the same as the distance of V_7 from the new source U_5.

5.2.3 The algorithm

Definitions: The *state* of a process x is a 4-tuple (pref, aug, choice, dist); *pref* (short for preference) of a process can be null or p in $N(x)$; *aug*(short for augment) of a process is s(silent), r(requesting), a(acknowledging) or c(changing); *choice* of a process can be null or y in $N(x)$; *dist*(short for distance) of a process can be any non-negative integer less than or equal to $N-2$. If process x has $x.\text{aug} = r$, then process x is *requesting*. The terms *silent*, *acknowledging* and *changing* are defined similarly. If a process x has $x.\text{dist} = d$ then process 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. Each process x can determine for each of its neighbours y if y's pref (or choice) is null, points to x, or points elsewhere.

Let $\widehat{N}(x)$ denote the set $N(x) - \{x.\text{pref}\}$. A process's *state is valid* if it is in state u.1 through u.6 or v.1 through v.5 in Table 5.1 and its choice is null or in $\widehat{N}(x)$; otherwise it is *invalid*.

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

To simplify the description of the algorithm, child and parent relationships are defined for processes 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 than this only because adjacent processes have various combinations of states in the course of an augmentation process.

For a process $u \in U$ 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 process u, denoted by $mac(u)$, is defined as

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

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

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

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

Let $u \in U$ be a process with $u.\text{aug} = a$, then

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

Let $v \in V$ be a process with $v.\text{aug} = a$, then

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

Let $u \in U$ be a process with $u.\text{aug} = r$ or $u.\text{aug} = a$, then

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

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

Table 5.1: Consistency predicates for valid states

Let $v \in V$ be a process with $v.\text{aug} = r$ or $v.\text{aug} = a$, then

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

Observation 5.2.1. *If* child$(x) = y$ *for processes x and y, then* parent$(y) = x$. *If x and y are both acknowledging processes and* parent$(x) = y$, *then* child$(y) = x$.

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

Table 5.2: State transitions

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

Step 1 (State validity check): If process 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 5.1 is false, the process resets to a new state as given in Table 5.1.

Step 3 (Preference consistency check): If a process 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 process x makes the state transition in Table 5.2 corresponding to its state if and only if the guard for that state is true.

Figure 5.3: `Bipartite Maximum Matching Algorithm`

5.2.4 Analysis and correctness

The acknowledgment of a process x is *real* if either x is in partition V and in state v.2 (process x is called a *sink*), or child$(x) = y$ and y's acknowledgment is real. If a process's acknowledgment is not real then it is *spurious*.

The predicate P is defined over the configurations of the system as follows. For any configuration c, $P(c)$ is true if and only if, in c, the system is in a maximum matching and no process has a spurious acknowledgment.

We will prove that the `Bipartite Maximum Matching Algorithm` is stabilizing, given that the processes know their bipartition.

Overview of the proof: The proof of correctness follows the conventional safety and progress method. Since the algorithm is self-stabilizing, the processes can start in any arbitrary state. A process can detect the validity of its state locally. If a process starts in an invalid state, it is detected in the first round and the process resets to state u.1 or v.1 depending on its partition. Lemma 5.2.2 ensures that once a process is in a valid state, it is always in a valid state.

The matching of the underlying graph is determined by the states of the processes. Lemma 5.2.3 ensures that the matching determined by the processes' states is a matching of the underlying graph.

The safety property is that the size of the matching of the system never decreases. This is established by Corollary 5.2.8, which uses Lemma 5.2.6 and Lemma 5.2.7.

The progress property of the system is established by proving that as long as the matching of the system is not maximum, an augmenting path is detected by the processes and an augmentation along that path subsequently increases the size of the matching. This is proved in several steps.

Recall that an acknowledgment is initiated only by an unmatched V process, referred to as a sink. This acknowledgment is then copied back to the source by other processes on the augmenting path. Such a path along which an acknowledgment is copied, is called an acknowledging path. However, since the algorithm is a self-stabilizing algorithm, a process may start as an acknowledging process, even though the acknowledgment is not associated with a sink. This acknowledgment, referred to as a spurious acknowledgment, may get copied to other processes along an alternating path. Since each process has knowledge only about its neighbourhood, only the process at the very end of this path can detect the spurious nature of its acknowledgment. Lemma 5.2.9 and Corollary 5.2.10 establishes that no new spurious acknowledgments are created and Lemma 5.2.11 ensures that if the system starts with some spurious acknowledgments, they disappear within N rounds. Hence after N rounds, if any process is acknowledging, then there is an acknowledging path from that process to a sink.

While finding an augmenting path, a requesting tree is built, rooted at a source. Lemma 5.2.12 establishes that the path from the source to the sink, determined by the parent pointers, is an augmenting path based on the current matching. Lemma 5.2.13 ensures that once an augmenting path is detected, some augmenting path is reserved all the way to the source within a bounded number of rounds. Lemma 5.2.14 ensures that once an augmenting path is reserved from sink to source, the augmentation process is completed within a bounded number of rounds and the size of the matching increases.

Due to the pruning process, a non source node can become the root of a pruned subtree. No process in that subtree participates in the current augmentation process. Thus, a requesting process in that subtree has a spurious request and an acknowledging process is an orphan, meaning that they are no longer associated with a source. Spurious requests may obstruct progress by preventing a real request from being propagated. Lemma 5.2.16 establishes that if new spurious requests are created (this can happen only due to a pruning process) then there exists an augmenting path that augments the matching within a bounded number of rounds. Lemma 5.2.17 establishes that if no new spurious request is created, the existing spurious requests either disappear, or attach themselves to a requesting tree and become real within a bounded number of rounds. Similarly, Lemma 5.2.15 establishes that if new orphan acknowledgments are created (again, this can happen only due to a pruning process) then there exists an augmenting path that augments the matching within a bounded number of rounds. Lemma 5.2.18 establishes that if no new orphan acknowledgment is created, the existing orphan acknowledgments either become silent or attach themselves to a requesting tree and become non-orphan within a bounded number of rounds.

Finally, progress of the algorithm is established by Lemma 5.2.19. This lemma establishes that as long as the matching is not maximum, an augmenting path is detected and an augmentation along that path increases the size of the matching within a bounded number of rounds. Stability of the maximum matching is established by Lemma 5.2.20.

Recall the definition of a valid state, given in subsection 5.2.3.

Lemma 5.2.2. *After the initial round, each process is always in some valid state.*

Proof. Any process 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. Process 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 5.2.2 implies that Step 1 is a no-op after the first round. The reminder of the proof applies after the first round.

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

Lemma 5.2.3. *The set M is a matching.*

Proof. A process 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 process x is *matched* if $xy \in M$ for some process y, and x is *matched with* y if $xy \in M$. If a process is not matched, then it is *unmatched*.

Observation 5.2.4. *A process does not change its preference in Step 2.*

Observation 5.2.5. *If a process is matched or its preference is null, then the condition in Step 3 is false and therefore Step 3 is a no-op.*

Lemma 5.2.6. *A U process does not change its preference while it is matched.*

Proof. By Observations 5.2.4 and 5.2.5, the only way a matched U process can change its preference is in Step 4. A non-null preference changes in Step 4 only in Row 5 of Table 2, when a process is not matched. □

Lemma 5.2.7. *A matched V process never becomes unmatched.*

Proof. By Lemma 5.2.6, a U process never changes its preference while matched. Hence the only way a matched V process may become unmatched is if it changes its own preference. Because no process changes its preference in Step 2, and Step 3 is no-op for a matched process, the only way a matched V process can change its preference is in Step 4, by executing Row 11 of Table 5.2. In this case it is immediately matched with its choice. □

Corollary 5.2.8. *The cardinality of M never decreases.*

Recall the definition of a real acknowledgment, given at the beginning of this subsection. Informally, a real acknowledgment refers to an acknowledging process that is on an alternating path that is reserved all the way to an unmatched V process (a sink). If x_1 is an acknowledging process 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, processes $x_2, \ldots, x_{m-1}, (m > 3)$, are matched.

Lemma 5.2.9. *If $x_1, \ldots, x_m, (m \geq 2)$ is an acknowledging path leading to sink x_m, then no process on this path can change state before x_1 changes.*

Proof. Since x_m has zero preference, by Observation 5.2.5, Step 3 is no-op for x_m. The state consistency predicate for x_m (Row v.2 in Table 5.1) is true. The guard in Row 8 in Table 5.2 is false for x_m as long as x_{m-1} is acknowledging. Hence x_m can not change state before x_{m-1}. This proves the lemma for $m = 2$.

Let $m > 2$. 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 5.2.1 each also has a non-zero parent. Therefore the state consistency predicate is true and Step 2 is a no-op for $x_i, 2 \leq i \leq m-1$.

Since $x_i, 2 \leq i \leq m-1$, is matched, Step 3 is no-op for $x_i, 2 \leq i \leq m-1$, by Observation 5.2.5.

Each U process in the path $x_i, 2 \leq i \leq m-1$, is in state u.5 of Table 5.1 and is matched. Therefore its guard in Row 5 of Table 5.2 is false. Hence no U process in this path can make a state change before x_1.

Any V process, v, other than x_m, on the acknowledging path is in state v.5 of Table 5.1. Let parent$(v) = v$.choice $= u$. Then u must be a U process on the acknowledging path. For the guard in Row 11 of Table 5.2 to be true, u.aug $= c$. But no U process on the acknowledging path has changed state. So the guard in Row 11 of Table 5.2 false. Hence no V process on the acknowledging path can change state before x_1. □

Corollary 5.2.10. *After the initial round, a real acknowledgment never becomes spurious.*

Proof. By Lemma 5.2.2 any process v in V with v.pref $= 0$ and v.aug $= a$ must be in state v.2, which is, by definition, real.

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

Lemma 5.2.11. *After N rounds, no process has a spurious acknowledgment.*

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

By Lemma 5.2.2, after the initial round, any acknowledging process with zero distance must be in state v.2, which is by definition real.

Suppose that after $k \geq 1$ rounds, any acknowledging process x with x.dist $\leq k-1$ has a real acknowledgment. Let x be an acknowledging process with x.dist $= k$ at the start of round $(k+1)$.

By definition, child$(x) = 0$ or child$(x) = y$ for some acknowledging process y with y.dist $= k-1$. If child$(x) = 0$, then the state consistency predicate (states u.5 or v.5 in Table 5.1) 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 process 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 process copies an acknowledgment with distance $\geq N-2$, the lemma follows. $\square$

Let x be a requesting, acknowledging or changing process. If parent$(x) = 0$, then *root*$(x) = x$. If parent$(x) \neq 0$, then there exist processes $x_1, \ldots, x_m$ 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 *root*$(x) = x_m$.

A process $u \in U$ 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 5.2.12. *If u is a source and v is a sink such that* root$(v) = u$*, then the 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 every process on this path.*

Proof. If $m = 0$, then u and v are two adjacent unmatched processes. 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 process $u_i v_i \in M$. Hence Q is an augmenting path.

Every process x on Q, except for the source, has parent$(x) \neq 0$. If x and y are two processes 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 processes on Q. $\square$

Let x_1 be a process 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 process $x_i, 2 \leq i \leq m-1$, is matched.

Let $Q = v, x_1, \ldots, x_m, u$ with parent$(v) = x_1$, parent$(x_i) = x_{i+1}, 1 \leq i \leq m-1$, and parent$(x_m) = u$ 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.aug $\neq r$ then Q is called a *fully acknowledged augmenting path* or an *FA-path*.

If there exist processes x_i and x_{i+1} (or u) on this PA-path such that x_i.aug $= a$ and x_{i+1}.aug $= r$ (or u.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 processes 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 5.2.13 and 5.2.14.

Lemma 5.2.13. *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 is 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

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

where u is a source with u.aug $= r$, and $x_m, \ldots, x_1, v$ is an acknowledging path with parent$(x_m) = u$. Hence the guard in Row 2 of Table 5.2 is true for u, and after u executes Row 2, Q is an FA-path.

Assume that the result is true for some $k \geq 1$ and suppose that $R = v, x_1, \ldots, x_m, u$ is a PA-path with source u whose unacknowledged part has size $(k+1)$; i.e, u.aug $= r$ and x_i.aug $= r$, $m-k+1 \leq i \leq m$. By Lemma 5.2.9 the only process that can change state on the path $x_{m-k}, \ldots, x_1, v$ is x_{m-k}. But x_{m-k} cannot change state before parent$(x_{m-k}) = x_{m-k+1}$ changes state.

By Lemma 5.2.12, R is an augmenting path, implying that $x_{2i} \in V, 1 \leq i \leq m/2$, and $x_{2i-1}x_{2i} \in M$. The only possible state change for a requesting V process $x_i, m-k+1 \leq i \leq m$ on R is Row 10 of Table 5.2. But this requires x_{i-1}.aug $= a$, which is only possible if $i = m-k+1$. If x_{m-k+1} changes state, the path R is a PA-path whose unacknowledged part has size k.

The possible state changes for a requesting U process on R are Row 2 of Table 5.2, for source u, and Row 4 of Table 5.2, for any other process $x_i, m-k+1 \leq i < m$. In either case there exists a V process v with a real acknowledgment such that parent$(v) = u$ or parent$(v) = x_i$. Since v has a real acknowledgment (Lemma 5.2.11), there is an acknowledging path $v, y_1, \ldots, y_l, l \geq 1$. If u executes Row 2, we have an FA-path $u, v, y_1, \ldots, y_l$. Otherwise, if x_i executes Row 4, there is a PA-path $u, x_m, \ldots, x_i, v, y_1, \ldots, y_l$ whose unacknowledged part has size at most k. Hence the result follows by induction. ☐

Lemma 5.2.14. *After N rounds, if there is an FA-path, then the size of the matching increases 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.aug $= a$ then the guard in Row 8 is false for v and guard in Row 5 is true for u. Hence after the first round, u.aug $= c$ making the guard in Row 8 true for v. During the second round, v executes Row 8, and u and v are matched with each other. Since 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.aug $= a$, then by definition of source, the guard in Row 5 is true for u, and in the next round, u.aug $= c$. If u.aug $= c$ then, since parent$(x_0) = u$, the guard in Row 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 increases 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 process becomes an orphan (defined below), a consequence of the pruning process. A second change that ensures the existence of a PA-path is the creation of a spurious request (defined later), another consequence of the pruning process.

An acknowledging process x is an *orphan* if its acknowledgment is real, root$(x) = y$ and y is a not a source.

Lemma 5.2.15. *After N rounds, if an acknowledging process becomes an orphan, then there exists a PA-path.*

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

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

where x_m is a sink and $\text{root}(x_l) = x_1$ is a source. If x_l becomes an orphan, then some process $x_i, 1 \leq i \leq l-1$, 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 $x_2, \ldots, x_m$ is an acknowledging path to a sink x_m, Lemma 5.2.9 guarantees that the only process on this path that can change state is x_2. The only possible state change for x_2 is Row 11, making x_3 a source. Thus x_l does not become an orphan.

If x_1 is in state u.5, then $x_1, \ldots, x_m$ is an acknowledging path to a sink x_m. Again, Lemma 5.2.9 ensures that the only process on this path that can change state is x_1. The only possible state change for x_1 is Row 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 $k, 1 \leq k \leq l-1$, $x_1, x_2, \ldots, x_k$ is a requesting path, and $x_{k+1}.\text{aug} = a$. Since $x_{k+1}, \ldots, x_m$ is an acknowledging path to a sink x_m, Lemma 5.2.9 ensures that only process 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 Row 10. This state change does not alter the parent value of any process on Q. If $x_k \in U$, then the possible state changes are Row 2 or Row 4. In either case the execution of these lines requires a process $v \in V$ with $\text{parent}(v) = x_k$. Since v's acknowledgment is real (Lemma 5.2.11), 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. $\square$

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

Lemma 5.2.16. *After N rounds, if the request of a process changes from real to spurious, then there exists a PA-path.*

Proof. By Lemma 5.2.2, after the first round, any requesting process is in state u.2, u.4 or v.4. If process is in state u.2, then it is real and cannot be spurious.

Suppose process 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 process 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 process, x_i.pref $= x_{i-1}$. The only state change possible for x_i (state v.4) is Row 10, and this requires x_{i-1}.aug $= a$. But x_{i-1}.aug $= r$. Hence x_i can not be in partition V. The possible state changes for a requesting U process are either Row 2 for process x_m in state u.2, or Row 4 for process x_i $(i < m)$ in state u.4. In either case, there exists an acknowledging V process v with parent$(v) = x_i$. Since v's acknowledgment is real (Lemma 5.2.11), 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. □

Lemma 5.2.17. *If no process's request changes from real to spurious in rounds r through $(r+l)$ inclusive, $r > N, l > N$, then there is no spurious request in rounds $(r+N)$ to $(r+l)$ inclusive.*

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

By Lemma 5.2.2, after round r, any requesting process 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 process x with x.dist $\leq k$ has a real request. Let x be a requesting process with x.dist $= k+1$, at the start or round $(r+k+1)$. By definition, parent$(x) = 0$, or parent$(x) = y$ for some requesting process y with y.dist $= k$. If parent$(x) = 0$, then the state consistency predicate (state u.4 or v.4 in Table 5.1) for x is false, triggering a reset to x.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 process 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 process copies a request with distance $\geq N-2$, the lemma follows. □

Lemma 5.2.18. *After N rounds, if there is no spurious request for n consecutive rounds, then either all orphaned processes become silent or there exists a PA-path.*

Proof. Suppose x_i is an orphaned process, 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 processes 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 process 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 processes become silent or there exists a PA-path. □

Lemma 5.2.19. *After N rounds, if the matching is not maximum, then its cardinality increases 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 process changes from real to spurious, or any acknowledging process becomes orphaned, then by Lemma 5.2.16 or Lemma 5.2.15 respectively, there is a PA-path.

Otherwise, during I, no process's request changes from real to spurious and no process becomes orphaned. By Lemma 5.2.17, after round $(r+N-1)$, there are no spurious requests. Thus by Lemma 5.2.18, either there is a PA-path, or there are no orphaned acknowledging processes 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 process, a request propagates 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 5.2.13 and Lemma 5.2.14 that the cardinality of M increases within another $2n$ rounds. □

Lemma 5.2.20. *After N rounds, if the matching is maximum, then no matched process changes its preference.*

Proof. Suppose M is a maximum matching and $uv \in M$. By Lemma 5.2.6, 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 Row 11. In this case v is in state v.5 and by Lemma 5.2.11 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 process u_0 in state u.6, such that v.choice $= u_0$ and u_0.pref $= v$ (refer to the guard in Row 11). Hence parent$(v) = u_0$, and therefore $u_0, v, x_1, \ldots, x_m$ is an FA-path. By Lemma 5.2.12, this is an augmenting path, contradicting the assumption that M is maximum. Therefore the matching does not change. □

Lemmas 5.2.19 and Corollary 5.2.8 combine to prove the main result.

Theorem 5.2.21. *Starting from an arbitrary configuration, the system converges to a maximum matching within $O(Nn)$ rounds.*

Closure: Lemma 5.2.11, Corollary 5.2.8 and Lemma 5.2.20 together prove that once the system is in a legitimate configuration, it always remains in a legitimate configuration.

Convergence: Recall that in a legitimate configuration of the system, no process has a spurious acknowledgment and the system is in a maximum matching. Lemma 5.2.11 establishes that the there are no spurious acknowledgment in the system within a bounded number of rounds. Theorem 5.2.21 establishes that the system converges to a maximum matching within a bounded number of rounds. Together they establish that starting from an arbitrary configuration, the system converges to a legitimate configuration within finite amount of computation.

5.3 Fair Composition

We proved that the `2-colouring Algorithm` is self-stabilizing and the `Bipartite Maximum Matching Algorithm` is stabilizing, given that the processes know their bipartition. The `2-colouring Algorithm` does not use the variables of the `Bipartite Maximum Matching Algorithm`. The `Bipartite Maximum Matching Algorithm` only reads the partition information of the `2-colouring Algorithm`. Hence a fair composition of these two algorithm solves the bipartite maximum matching problem in an arbitrary bipartite network with distinct identifiers. The combined algorithm takes $\mathrm{O}(Nn)$ cycles (recall that the round complexity and cycle complexity are the same for composite atomic algorithms) to converge to a legitimate configuration in the worst case.

Chapter 6

Comments and Future Work

For the sake of clarity, we assumed the locally shared memory model for the maximal and the maximum matching algorithms. In the case of link register model, a process has to copy its shared memory contents to all the link registers. In the case of the composite atomic model, this operation can be done in one atomic step. Hence very little modification is required in the algorithm and no modification is needed for the correctness proof. In the case of read/write atomic model, the process has to copy the shared memory contents to one link register at a time. We are currently working to prove that even in this case, the algorithm works correctly.

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. The maximal matching algorithm does not require any global information about network topology. Hence in this case, a network topology change is equivalent to a system configuration change and the self-stabilizing property of the system ensures that the system re-converges to a legitimate configuration within finite amount of computation. The maximum matching algorithm needs an upper bound on the network size. As long as the topology change does not violate this upper bound, a network topology change is again equivalent to a system configuration change.

The following sections describe the issues relevant to each problem.

6.1 Maximal Matching

Maximal matching is a local property in a graph. Therefore, processes need only local knowledge to solve this problem. Ideally, a self-stabilizing algorithm should need only a constant number of cycles to converge. But the `Maximal matching algorithm` finds the particular maximal matching defined by the graph theoretic scheme, presented in subsection 4.3.2. To determine this particular maximal matching, processes may need to propagate information throughout the entire network and hence may need a linear number of cycles to converge.

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

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 chosen from a large set and are assigned in an unfavourable manner. However, this cannot happen if the labels are assigned using the label generation algorithm and the maximum degree of the network is constant.

In general, it is difficult to design and analyze read/write atomic algorithms. Since processes are forced to take decisions based on outdated information, the scheduler has a great deal of extra power to avoid convergence beyond that power available for composite atomic systems. We present an obvious extension of the Hsu-Huang algorithm in Appendix A, using randomization. We prove that the algorithm works for trees under read/write atomicity. This algorithm has the local re-convergence property, meaning that a local fault is contained and corrected locally. This algorithm does not work in a general network. We are currently working on finding the step complexity of the `Maximal Matching Algorithm`.

The label generation algorithm can be used for other applications. A self-stabilizing algorithm to determine a maximal independent set in a graph is presented in Appendix B. This algorithm needs locally distinct labels that can be generated using the label generation algorithm.

6.2 Maximum Matching

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 [15, pp. 226-243] to overcome this problem in a sequential algorithm. Implementing this technique in a distributed setting poses challenges. Our future work will be directed towards solving these problems.

Bibliography

[1] A. Arora and M.G. Gouda. Closure and convergence: a foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19:1015–1027, 1993.

[2] H. Attiya and J. Welch. *Distributed computing: fundamentals, simulations and advanced topics*. The Mc-Graw Hill Companies, 1998.

[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] J.R. Griggs and Y.K. Roger. Labelling graphs with a condition at distance 2. *SIAM J. on Discrete Mathematics*, 5:586–95, 1992.

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

[10] L. Higham. Simple randomized leader election with extensions. Technical Report TR90/407/31, Department of Computer Science, University of Calgary, 1990.

[11] 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.

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

[13] 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.

[14] N. A. Lynch. *Distributed algorithms*. Morgan Kaufmann Publishers, 1996.

[15] C.H. Papadimitriou and K. Steiglitz. *Combinatorial optimization: algorithms and complexity*. Prentice-Hall, 1982.

[16] M. Schneider. Self-stabilization. *ACM Computing Surveys*, 25:45–67, 1993.

[17] S. Shukla, D. Rosenkrantz, and S. Ravi. Developing self-stabilizing coloring algorithms via systematic randomization. In *Proceedings of the International Workshop on Parallel Processing*, pages 668–673, Bangalore, India, 1994. Tata-McGrawhill, New Delhi.

[18] S. Shukla, D. Rosenkrantz, and S. Ravi. Observations on self-stabilizing graph algorithms for anonymous networks. In *Proceedings of the Second Workshop on Self-Stabilizing Systems*, pages 7.1–7.15, 1995.

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

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

[21] D.B. West. *Introduction to graph theory*. Prentice-Hall, 2001.

Appendix A

Maximal Matching for Trees

A.1 Maximal Matching for Trees

A.1.1 Model

The network is an anonymous, arbitrary tree. The model of computation is locally shared memory and a distributed scheduler under read/write and fine atomicity.

The name of a processes, used in the following subsections, is for description purposes only. These names are not available to the processes.

A.1.2 Algorithm for processes x

Processes x has a shared variable ptr, that is either `null`, denoted 0, or points to a processes in $N(x)$. Each processes x can determine, for each of its neighbours y, if y's ptr is null, points to x, or points elsewhere. Function `rand`(x) returns a pointer to a random processes, chosen uniformly from $N(x)$. Local variable temp holds a ptr. The pseudocode of the algorithm is provided in Figure A.1.

A.2 Proof of correctness

A non-leaf processes is a **pendant** if it has at most one non-leaf neighbour. Processes x and y are *matched*, it x.ptr $= y$ and y.ptr $= x$ and neither of their pointers ever subsequently changes.

```
do forever
if x.ptr = 0 then
    if there exists y ∈ N(x) such that y.ptr = x then
        x.ptr ← y
    else
        temp ← rand(x)
        if temp.ptr ∈ {0, x} then
            x.ptr ← temp
        end if
    end if
else
    if (x.ptr).ptr ∉ {0, x} then
        x.ptr ← 0
    end if
end if
end forever
```

Figure A.1: `Maximal Matching for Trees`

Lemma A.2.1. *In any tree, if there is a non leaf node, then there is a pendant node.*

Proof. Let T be any tree with at least one non-leaf node. Create a new graph F by deleting all the leaf nodes and their adjacent edges of T. Since T has at least one non leaf node, F must have at least one node. Since T does not have any cycle and we have not added any edge, F must be a forest. Hence F must have a component with at least one leaf node, say x. In F the degree of x is 1. Hence in T, x has at most one non-leaf neighbour. Hence x is a pendant in T. $\square$

The program counter of a processes x, denoted $x.pc$, represents the line of the algorithm, processes x executes next.

Lemma A.2.2. *If a processes x points to a leaf y, then x never changes its pointer and x and y get matched within one cycle.*

Proof. Since y has no other neighbour other than x, y.ptr can only be x or 0. Hence the condition in line 11 is never true for x. Hence condition $x.\text{ptr} = y$ holds forever. If $y.\text{ptr} = x$ and $y.pc \leq 11$ then condition in line 11 is never true for y and x and y are matched. In other case, $y.\text{ptr} = 0$. After that, whenever the scheduler schedules y, it executes line 2. Then processes y finds the condition in line 2 is true for processes x. Consequently processes x and y become matched in one cycle. $\square$

Lemma A.2.3. *If a pendant processes executes line 3, there will be a match within one cycle.*

Proof. If the pendant processes x executes line 3 and points to leaf, then by Lemma A.2.2 they become matched within one cycle. If it points to a non-leaf processes y, then it must have executed line 2 before and read $y.\text{ptr} = x$. At that time $y.pc \neq 12$, because in that case $x.\text{ptr} = z$ at some earlier time, when processes y executed line 11. Since z is a leaf node, by Lemma A.2.2, processes x and z get matched and processes x never executes line 3, a contradiction. Hence $y.pc \leq 11$. Therefore, the condition in line 11 is never true for processes x and y and they are matched. $\square$

Lemma A.2.4. *With probability 1/2 a pendant processes becomes matched within four cycles.*

Proof. Within two cycles, a pendant processes either gets matched or drops its pointer. In the second case, if it executes line 3, then by Lemma A.2.3 it gets matched within one cycle. Otherwise it executes line 5, and with probability at least 1/2, it chooses a leaf node. After that, when it executes line 6, it finds the condition true and points to the leaf. Then by Lemma A.2.2, it becomes matched within one cycle. $\square$

Theorem A.2.5. *Starting from an illegitimate configuration, the system converges to a legitimate configuration within expected O(n) cycles, where n is the number of processes in the system.*

Proof. If the graph is just an edge then the two processes get matched trivially. In any other case, by Lemma A.2.1, there is a pendant processes, say x. By Lemma A.2.4, within expected 8 cycles, x becomes matched. After that, no other processes ever points to x and the node it is matched with. This essentially leads to a smaller system excluding x, the node it is matched with, and its neighbouring leaves, containing at most $n - 2$ nodes. The theorem now follows by induction on the number of processes. $\square$

Appendix B

Maximal Independent Set

Definition: An *independent set* of graph G with node set V and edge set E, is a set $U \subseteq V$, such that no two nodes in U are adjacent. An independent set is *maximal*, if no independent set properly contains it.

A network has *locally distinct labels*, if any two adjacent processes have different labels.

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

Algorithm for process x: Process x has a shared variable called Σ that takes a boolean value. It also has a boolean local variable L.

Proof of correctness: The maximal independent set of the system is the set of processes with $\Sigma = \texttt{true}$.

Process x is a *local minimum*, if for all process $y \in N(x)$, $x.\text{lid} < y.\text{lid}$.

Theorem B.0.6. *The system converges to a maximal independent set within n cycles, where n is the number of processes in the system.*

```
do forever
   L ← true
   for all y ∈ N(x)
      if y.lid < x.lid and y.Σ = true
         L ← false
   Σ ← L
```

Figure B.1: `Maximal Independent Set`

Proof. After the first cycle, a local minimum sets its Σ variables to `true` and never changes. After the second cycle, each neighbour of a local minimum sets its Σ variable to `false` and never changes. The algorithm now proceeds as if the local minima and their neighbours are removed from the system, resulting a new system containing at most $n-2$ processes. It follows by induction on the number of processes, that the system converges to maximal independent set in at most n cycles. □

Observe that the `Maximal Independent Set` algorithm needs locally distinct labels. A fair composition with the `Label generation algorithm` solves the Maximal Independent Set problem in an anonymous general network. However, the `Maximal Independent Set` algorithm needs only locally distinct labels, as opposed to locally distinct labels within distance 2. Hence a simpler algorithm to generate locally distinct labels will suffice.