THE UNIVERSITY OF CALGARY

Self-stabilizing Minimum Spanning Tree Construction on Message-Passing Network

by

Zhiying Liang

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

September, 2001

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "Self-stabilizing Minimum Spanning Tree Construction on Message-Passing Network" submitted by Zhiying in partial fulfillment of the requirements for the degree of Master of Science.

Supervisor, Dr. Lisa Higham,

Department of Computer Science.

Dr. Claudio Costi,

Department of Computer Science.

Dr. Karen Seyffarth ,

Department of Mathematics and Statistics.

Date

Abstract

Self-stabilization is an abstraction of fault tolerance for transient faults. It guarantees that the system will eventually reach a legitimate configuration when started from an arbitrary initial configuration.

This thesis presents two minimum spanning tree algorithms designed directly for deterministic, message-passing networks. The first converts an arbitrary spanning tree to a minimum one; the second is a fully self-stabilizing construction. The algorithms assume distinct identifiers and reliable fifo message passing, but do not rely on a root or synchrony. Also, processors have a safe time-out mechanism (the minimum assumption necessary for a solution to exist.) Both algorithms apply to networks that can change dynamically.

Acknowledgments

First of all, I would like to thank my supervisor Lisa Higham. I thank her for getting me interested in Distributed Computing, and her constant guidance, encouragement and support.

I also want to thank the constant support from my family, especially, my husband Michael Dery and my mother Jingfen Yu.

Finally, I acknowledge the support from the Department of Computer Science.

Contents

CHAPTER 1

Introduction

1.1 Models of Distributed Computing

The term *distributed system* is used to describe a communication network, a multiprocessor computer or a multitasking single computer. A distributed system contains of two types of components: *processors* and *communication channels* between the processors. *Distributed computing* studies the computational activities performed on these systems.

Distributed systems, by their very nature, are subject to much uncertainty. Uncertainty may be a result of failures: individual processors may crash, information may be corrupted, lost or replicated during transmission, processors may lose synchrony, and so forth. Therefore a major concern in the study of distributed computing is dealing with uncertainty. *Fault-tolerant* computing is geared towards the study of reliable distributed systems that tolerate uncertainty.

In the theory of distributed computing, one usually uses the term *model* to denote an abstract representation of a distributed system. An *algorithm* is the program given to the processors to solve a certain problem on a certain model setting. *Complexity analysis* provides some measurement of the performance of algorithm. For example, the problem can be mutual exclusion, leader election, or construction of a minimum spanning tree. The model can be shared memory or messaging-passing. The algorithm can be deterministic or randomized. The message complexity of the algorithm would provide a bound on the total number of message exchanged while the system solves the problem. The theory of distributed computing aims to study problems and their complexity measures under a variety of models.

Various different models arise depending on assumptions about how both processors and communication behave. The following subsections describe some common alternative choices for several different components of a model of a distributed system.

1.1.1 Communication

The communication model describes the mechanism that supports information exchange between processors. Two common interprocessor communication models are the *message passing* model and the *shared memory* model.

In the message passing model, processors communicate by exchanging messages. A processor sends a message by adding it to its outgoing message queue, and receives a message by removing it from its incoming queues. Different sub-models arise from different assumption about both channel and queue behavior. Queues may have bounded or unbounded size. If they are bounded then the model must specify what happens when a message is added to a full queue. Typically, it is lost or some other message in the queue is lost. Generally bounded queues are more realistic models of channel behavior.

Each processor may have one queue for the entire set of all incoming messages or it may have a separate queue for each adjacent communication channel. The model may include the capability to send one message to a specified neighbor in one step, or broadcast a single message to any subset of neighbors, or even send different messages to each neighbor in one step.

A communication link is either *unidirectional* or *bidirectional*. A unidirectional communication link from processor P_1 to processor P_2 is used to transfer messages from P_1 to P_2. A unidirectional communication link can be described as a FIFO queue containing all the messages send by processor P_1 to its neighbor processor P_2 and not yet received by P_2. When P_1 sends a message to P_2, the message is added to the end of this queue. P_2 may receives a message m from the head of the queue. At the same time the message m is removed from this queue. A bidirectional communication link between P_1 and P_2 is modelled by a pair of FIFO queues, one in each direction.

In the shared memory model, processors communicate through globally shared objects. Typically these objects are atomic registers. An atomic register is a shared variable that can be either read or written in one indivisible (atomic) step. An atomic register can be multi-reader/multi-writer, or multi-reader/singer-writer, or single-reader/single-writer. But sometime the model includes stronger objects such as test-and-set, or fetch-and-add and other read-modify-write objects such as queues and stacks. Processors may write in a set of registers and may read from a possibly different set of registers. Most of the techniques used in the shared memory setting can be adapted for use in the network setting.

The link register model is a special restriction of the shared memory model. The network is modelled as a graph. However, rather than the message passing model, in the link register model, there are two registers between each pair of neighboring processors. A processor P_1 communicates with its neighbor processor P_2 by writing in the register r_{12} and reading from the register r_{21}. The processor P_2 communicates with P_1 by writing in r_{21} and reading from r_{12}. The link register model is commonly used in the self-stabilizing setting.

Of course these different assumptions impact the cost of algorithm solutions to distributed computing problems. They also differ in how accurately they reflect the real system they are intended to model.

1.1.2 Timing model

The three basic models of timing in distributed systems are called the *Synchronous* model, the *Asynchronous* model and the *Partially Synchronous* model.

The synchronous model, where each processor simultaneously executes one step of its program in each time step, is the simplest model to describe, to program and to analyze. Understanding first how to solve a problem in the synchronous setting is often useful for developing the algorithms to solve the same problem under different more realistic timing assumptions. But sometime it is impossible or inefficient to implement the synchronous model in a distributing system.

In the asynchronous model, processors execute their programs at different speeds. Both the absolute speed of each processor and the relative speed between processors may vary arbitrarily during the computation. The asynchronous model is harder to program than the synchronous model. Without timing restrictions, problems are more general and interesting and more realistic. On the other hand, it is typically harder to solve problems efficiently when there are no timing guarantees. Many problems that are solvable under the assumption of synchrony become impossible to solve in the asynchronous model.

The partially synchronous model assumes some restrictions on the relative timing, but execution is not in lock-step as it is in the synchronous model. It is a realistic model but often difficult to program. An algorithm tuned for one partial-synchrony assumption may fail for others. Thus these algorithms may lack portability. Violation of the timing assumption can cause the algorithm to fail to operate correctly.

A probability distribution on communication speeds, another kind of timing model, has been assumed in some literature. Although not many algorithms have been designed for models with probability distributions on communication speed, this model has an advantage and potential since it can be tuned for likely transmission speeds while maintaining correct behavior when timing behavior is unusual.

1.1.3 Scheduler

In an asynchronous system, these differences in the speeds of the processors are simulated with the use of a scheduler, alternatively called a *daemon*. It is assumed that at each time step a scheduler determines which processors execute the next step of their program subject only to synchronization enforced by explicit program constraints. The *distributed* daemon and the *central* daemon are two types of scheduler. In each step, the *central* daemon activates only one processor at a time. In each step, the *distributed* daemon selects a nonempty set of processors and activates all the processors in the set simultaneously. The central daemon is a special case of the distributed daemon in which the set of activated processors consists of exactly one processor. Some algorithms work correctly under a central daemon but not a distributed daemon. The requirement of a central daemon is an unreasonable constraint for a truly distributed system. In particular, its implementation requires some form of centralized control.

The scheduler is typically constrained by a fairness assumption that provides some minimum guarantee on the interval between successive steps of a processor. There are many different strengths of fairness. *Weak fairness* only ensures that in an infinite execution, each processors takes an infinite number of steps. *k-fairness* ensures that no processor executes more than k steps between any two successive steps of any other processor. A *round robin* scheduler constrains processors to take a fixed order under a 1-fairness assumption. The round-robin is the most restricted and often unrealistic scheduler. In this thesis, we assume the scheduler is only constrained to be weakly fair.

1.1.4 Atomicity

An *atomic step* is the largest step that is guaranteed to be executed uninterruptedly. *Composite* atomicity and *read/write* atomicity are two common kinds of atomicity. Composite atomicity allows a processor to perform several operations on shared memory in one atomic step. For example, under composite atomicity, a processor could read the state of each of its neighbors and update its own state as one indivisible action. A conservative assumption is that in a shared memory system an atomic step contains only a single read operation or a single write operation. This is called read/write atomicity.

Clearly an algorithm that is designed to work under read/write atomicity can be used in any system with composite atomicity. However, an algorithm that solves a problem under composite atomicity may be incorrect for read/write atomicity. Often it is easier to find an algorithm and prove it correct for composite atomicity than for read/write. This motivates one active research direction which is to design general technique for converting an algorithm correct for composite atomicity to one correct for read/write atomicity.

In randomized algorithms, there exist random operation such as coin tosses. If the random operation is not separable from the next read or write operation, then we call it *coarse atomicity*. The term *fine atomicity* means an atomic step contains only a random operation or a read operation or a write operation.

1.1.5 Network Topology

A message-passing system is generally modelled as a graph where processors are the nodes and the communication links are edges. In this case, the topology of the network is fixed. The assumptions about processors' knowledge of their network can have a significant impact on the network's ability or cost to solve various problems. For example, an algorithm can be designed for an arbitrary topology, or may be required to work only for a fixed family of topologies such as rings, trees, chains or complete graphs. As well, the algorithm may be parameterized by the network size, or it may be that the size is not available and can not be used explicitly by the algorithm. The direction of communication channels can be unidirectional or bidirectional. The network can be connected or unconnected, weighted or unweighted.

We may need to solve a problem only on a fixed specified graph, for example a mesh of 25 nodes. Sometimes, problems need to be solved on a family of similar graphs, for example, a bidirectional ring of unknown size or an arbitrary graph of known size. Nevertheless, the most interesting situations is when a problem is to be solved for an arbitrary graph. An even more challenging situation is when the network can change overtime and a solution must be designed so that it continuously updates under these dynamic changes.

1.1.6 Network Labels

Network labels play an important role in modelling the system. The *node label* and the *edge label* are two kinds of network labels.

A system may have totally identical processors in which processors (nodes) do not have unique identifiers (labels). The term *anonymous* system is used for such systems. Sometimes, a distinguished processor (node) in network has a unique identifier (label) and all the others have the same label. Most of papers we refer in Chapter 2 are modelled in this setting, regarding the node with the unique label as the root. In many cases, all the processors (nodes) in a system may have distinct identifiers (labels).

Sometimes, edges in network could have no labels, for example, the network could be anonymous. Sometimes, edges in network could have distinct labels, which could be represented by their end-point nodes if the network nodes have distinct identifiers. Sometimes, edges could be labeled to reflect some properties, for example, the weight of an edge could represent the cost of communication on the link. Sometimes, edges are partitioned into the different classes, labelled by distincted colors.

1.1.7 Fault Models

There are also different fault models in the context of fault-tolerant computing. For instance, we can consider four types of processor failures: stopping failures, where faulty processors can, at some point, just stop executing their local protocol, and crash/recovery failure, where processors can continuously execute their local protocol after recovering from their stopping failure, and crash/restart, where processors restart their local protocol after recovering, and Byzantine failures, where faulty processors can exhibit completely arbitrary behavior (subject to the limitation that they cannot corrupt portions of the system to which they have no access). We can also consider some communication failures: lost data, duplicated data, corrupted data and reordering of data during communication.

1.1.8 Algorithm Type

An algorithms may be *deterministic* or may employ *randomization*. There are two kinds of randomized algorithms, *Las Vegas randomized algorithms* and *Monte Carlo randomized algorithms*. A Las Vegas randomized algorithm solves a problem P if, for any instance of

P, it terminates with probability 1, and upon termination the solution is correct. A Monte Carlo randomized algorithm solves a problem P if, for any instance of P, it terminates within a bounded number of steps, and the solution is correct with probability at least p where $0 < p < 1$. Randomized algorithms are often used to break symmetry in an anonymous system.

If every processor in the system performs the same transition function, the algorithm is called *uniform*. In particular, if the system is anonymous and the network is regular, then all the processors are indistinguishable, so any algorithm for that system must be uniform. If one distinguished processor, often called the root or the leader, has a distinct transition function, and all others have the identical transition function, then the algorithm is called *semi-uniform*. Finally, if every transition function for each processor is distinct, then the algorithm is called *non-uniform*. A non-uniform algorithm is only possible if every processor is distinct, for example, in the system with distinct identifiers.

1.2 Self-stabilization

1.2.1 History and General Description

Large networks of processors are typically susceptible to transient faults and they are frequently changing dynamically. Ideally, basic primitives used by these systems can be made robust enough to withstand these faults and adapt to network changes. In 1974, Dijkstra [5] proposed a *self-stabilizing* model for achieving fault-tolerance in the presence of transient faults that can meet these requirements. According to Dijkstra, a system is self-stabilizing when "regardless of its initial state, it is guaranteed to arrive at a legitimate state in a finite number of steps." A system that is not self-stabilizing may stay in an illegitimate state forever. He observed that "The complication is that a node's behavior can only be influenced by the part of the total system state description that is available in that node: local actions taken on account of local information must accomplish a global objective". Dijkstra defined the *self-stabilizing* mutual exclusion problem on a ring of n finite state processors, where directly connected processors were called neighbors. His algorithm is semi-uniform because it requires the existence of a distinguished processor which behaves differently from the others. Dijkstra also introduced a central daemon in his model. In his paper, a processor is said to hold a token (Dijkstra called it a *privileged* machine) if a boolean function of the state of the processor and states of its neighbors is true. So, a processor can make a move by changing its state when it holds a token. A malicious scheduler picks one of the privileged processors, at each step, in an arbitrary but weakly fair order, to take its next move. In a move, a processor enters the new state, which is a function of its old state and the states of its neighbors. As the result of a move, the processor may become unprivileged, but its neighbour could become privileged.

The global configuration of the algorithm consists of the states of all n processors. The legal configurations are those configurations in which exactly one processor holds a token. The specification of a correct system also requires that the token be passed on the ring in a round-robin fashion. Dijkstra's *self-stabilizing* mutual exclusion algorithm guarantees that eventually the system will have a unique token to pass around the ring even when there are many tokens, or there is no token, initially present in the ring.

Although the concept of self-stabilization was proposed twenty five years ago, there were very few published results about self-stabilizing systems in the first fifteen years. In 1983, Lamport [13] said the following at his invited address to the 3rd ACM Symposium on Principles of Distributed Computing: "I regard this as Dijkstra's most brilliant work, at least, his most brilliant published paper. It's almost completely unknown. I regard it to be a milestone in work on fault tolerance" and suggested self-stabilization as a very important and very fertile field for research. In the years following Lamport's announcement, there has been a flurry of papers in self-stabilization. Dijkstra's notion of self-stabilization, which originally had a very narrow scope of application, is proving to become a unified method to resolve transient failures on distributed systems.

1.2.2 Definition of Self-stabilization

Before defining self-stabilization for a system, some preliminary definitions are introduced. Let S be a distributed system and let C be an arbitrary component of S. The *local state* of component C is the complete state description for C. The *global configuration* of distributed system S is the list of the local state of all the components in the system S. A set of *legitimate configurations* is a subset of all possible configurations, and represents those configurations in which the system S is supposed to be. In a legitimate configuration S is performing according to the design goals and in each step moves to another legitimate configuration barring any subsequent errors.

Transient faults cause processors to change their states resulting in an illegitimate global configuration. A *self-stabilizing* system will be able to overcome such a fault by eventually stabilizing to its required behavior without restarting the system. If the interval between two successive transient faults is long enough and if stabilization is fast enough, then the system should be able to converge to its correct behavior and achieve some useful work before the next fault causes the stabilization to repeat. More specifically, a self-stabilizing system guarantees that it will eventually reach the legitimate configurations when started from an arbitrary initial configuration. This behavior is called *Convergence*. Once a legitimate configurations is reached, the invariant is conserved for the rest of the execution of the protocol. This behavior is called *Closure*.

If a protocol is self-stabilizing the system need not be initialized, which can be a significant additional advantage especially for physically dispersed systems such the Internet.

1.3 Overview of Thesis

In this thesis, we present two distributed algorithms for the minimum spanning tree (MST) problem for a message-passing system on an arbitrary connected, weighted, undirected network where processors have distinct identifiers. Chapter 2 contains a survey of the literature related to self-stabilizing spanning tree and minimum-spanning tree construction.

Chapter 3 gives a precise description of the model. In Chapter 3, we motivate the ideas of our algorithm by outlining a new sequential algorithm that constructs an MST given an arbitrary spanning tree. While this algorithm is inefficient sequentially, it adapts well to a concurrent environment and is the motivation for our first distributed algorithm. Chapter 4 presents this algorithm, Basic_MST, and its correctness, and analysis. Although one main idea is contained in Basic_MST, substantial modifications and enhancements are needed to convert it to a general self-stabilizing MST algorithm. Chapter 5 presents the second algorithm, Self_Stabilizing_MST and its proof of correctness, and a complexity analysis. These algorithms are presented in an imaginary model where edges rather then nodes are processors. Chapter 6 shows a mapping technique for simulating the edge driven model by the usual network model. In Chapter 7, we summarize the contributions of this thesis, further comments and the future work.

CHAPTER 2

Literature Review

The book Self-Stabilization [6] written by Shlomi Dolev is the first and only book in the field of self-stabilizing distributed algorithms. Before I began my research on the minimum spanning tree problem, I studied the book and referred to it on many occasions. Even though the book was not intended to cover all the research activity in the field, I found it has quite extensive material for the beginner and the researcher. This Chapter reviews the concepts that will be used in the remainder of the thesis and the papers related to the self-stabilizing spanning tree problem.

Two important primitives for many protocols in distributed computing are construction of a spanning tree and construction of a minimum spanning tree. For example, a distributed message-passing network of processors might rely on an underlying spanning tree to manage communication. If the cost of using the different communication channels varies significantly, it may be desirable to identify the spanning tree with minimum cost. So a distributed self-stabilizing (minimum) spanning tree protocol would eventually converge to a global state where each processor identifies which of its adjacent edges are part of the required tree, regardless of what each processor had originally identified as part of the tree. As long as no further faults or changes occurred, this tree would remain unchanged once it was identified. Although there exist a number of self-stabilizing algorithms for the spanning tree problem [4, 11, 1, 8], only in [2] does the algorithm deal with constructing a minimum spanning tree (MST).

Let us first briefly review some previous works. Antonoiu and Srimani [1] presented a self-stabilizing distributed algorithm to construct an arbitrary (not necessarily breadth first) spanning tree. They say the algorithm has "a single uniform rule for all nodes in the graph". Because there is a distinctive root processor, however, it is non-uniform algorithm according to the definition of this thesis and as is commonly accepted. The algorithm runs under a distributed daemon and composite atomicity. Each processor maintains a pointer to its parent and a level value between 0 and $n-1$, which is its distance from the root. The root continuously sets its level to 0 and its pointer to itself. Every other node behaves as follows. The *legitimacy predicate* for node i is true when the parent of i is one of its neighbours and the level of i is 1 greater then the level of its parent. The algorithm proceeds in asynchronous rounds of communication where in each round the level and parent pointer of a node are updated if and only if the node's legitimacy predicate is not true. If no neighbour of node i has level less than i's level and i's level $< n$, then i increment its level, otherwise i assigns its pointer to any neighbour that has level, l, less than its own, and sets its own level to $l+1$. If the legitimacy predicate is true for every node, the configuration is legitimate, that is, it forms a spanning tree. The correctness that both pointer and level values eventually stabilize to a spanning tree depends on induction. In round one, the root must be correct. Assuming a subset of size k containing the root is correct, which means that the predicate is true for all the nodes in the subset, then this subset does not change its structure and, eventually, the correct subset increases by at least one node. The authors do not address the complexity of this algorithm. An upper bound, however, is easily achieved. Once a node is correct, it does not change its level or its parent pointer. In every round, the minimum level over all incorrect nodes is incremented until it is n or there are no incorrect nodes. Hence after n round all incorrect nodes have level n. Thus after at most an additional n rounds, the legitimacy predicate is satisfied by all the nodes. So the spanning tree is constructed in fewer than $2n$ rounds.

Antonoiu and Srimani [2] also proposed a distributed self-stabilizing algorithm for the minimum spanning tree problem. They claim that theirs is the very first paper on this problem in the self-stabilizing setting. The algorithm is designed for undirected connected and unique edge-weighted graphs with asynchronous, composite atomicity under the distributed daemon model. Every node in the system performs the same uniform rule. The rule is based on Maggs and Plotkin's results in 1988 [14] that consider the minimum spanning tree problem as a path finding problem. The main idea is to find ψ_{ij}, the minimum of the maximum of the weight of the edges in any path between two node i and j, for any nodes i and j. Maggs and Plotkin show that an edge e_{ij} is in the unique minimum spanning tree if and only if ψ_{ij} is equal to the weight of e_{ij}. For simplicity, Antonoiu and Srimani first introduced the rule and the proof for a particular node r. For a given r, every node i maintains two local variables, $L(i)$ and $D(i)$, containing information about the currently "best" path from i to r ("best" is intended to capture the path to r containing that edge with weight ψ_{ir}. Specifically, $L(i)$ is the *level* of node i, that is the number of edges on the current path and $D(i)$ is the maximum edge weight on the current path. So $D(i)$ is the current estimate of ψ_{ir}. The node i looks at the variables of its neighbours and takes action by updating its level $L(i)$ and its estimate $D(i)$. If all the neighbours have the reset value, which means the estimates at each neighbour of node i is wrong, then $L(i)$ and $D(i)$ gets reset as well. The "best" path to r from i, is via that neighbour j, such that the maximum of the weight of the edge to get to j and the weight of the maximum edge of the best path form j to r is as small as possible. Precisely $D(i) \leftarrow \min(\max_{j\in N(i)}(w(e_{ij}), D(j)))$. Since values may be initially incorrect, a reset is used if values are out of range, or other error are detected. The proof is again by induction and is similar to [1].

Chen, Yu and Huang [4] proposed a central daemon, composite atomicity and non-uniform self-stabilizing algorithm for constructing spanning trees. In their algorithm, each processor i, except the specific processor r called the root, maintains two local variables, *level* $L(i)$ and *parent* $D(i)$. The *level* is the estimated distance between i and r. The *parent* of i is a pointer to a neighbour of i. The level of r is always equal to 0 and has no parent pointer. Any other node i computes its variables by a set of rules. The rules are: if the parent p is in an error state, $L(p) = n$, then i must be in an error state too; if the parent has a level less than $n-1$, then assign the level plus one to $L(i)$; if i is in an error state and one of its neighbours is not, point to the neighbour as i's parent and assign the neighbour's level plus one to $L(i)$.

Even through the model is restricted, the most significant contribution of their paper is the approach of showing that a system is self-stabilizing. Their ideas are based on Kessels' model [12]. Let f be a function from a global configuration to a well founded set that satisfies:

- for every move of the system the value of f decreases.
- the value of f is minimum exactly for legitimate configurations.

In Kessels' approach a system is proven to be self-stabilizing by showing two things. First we show if the system has not stabilized, then some local predicate is true. This means that the corresponding processor can make a move. Second we show that such a function f exists. Chen, Yu and Huang partitioned all the nodes in the graph into the disconnected components of the "tree" edges in a configuration. For each component, the minimal level of the nodes in that component is determined. Let t_i be the number of components with minimum level i. They define the value of the configuration to be the vector $(t_0, t_1, \ldots, t_n)$. They prove that when these vectors are ordered lexicographically, then no matter what rule in the algorithm is applied, the vector value of the function decreases. Eventually, the value for the function must obtain $(1, 0, \ldots, 0)$, indicating that a spanning tree is formed. The system is self-stabilized. This proof technique is a significant contribution and provides a additional tool for correctness proofs.

Huang and Chen [11] also proposed a non-uniform self-stabilizing algorithm for constructing breadth-first trees. Distributed demon and composite atomicity are assumed in this paper. In their algorithm, except the specific processor r which is called the root, each processor i maintains two local variables, *level* $L(i)$ and *parent* $D(i)$. The *level* is the estimated distance between i and r. The *parent* of i is a pointer to a neighbour of i. The level of r is always equal to 0 and r has no parent pointer. For any other node i, if i's parent is not a neighbour of i with lowest level, then i updates its pointer to point to any such neighbour

j and sets $L(i) = L(j) + 1$. As they admitted, the model is also too restricted. However, their major contribution is providing a proving technique. Their ideas are similar to their previous paper [4]. Since a bounded function may be hard to find in many cases, in this paper, they transformed the original set of rules into another set of rules so that both sets of rules have an equivalent effect. Then, they found a bounded function for the transformed rules. They split one rule for two disjoint cases. In one case, function F_1 decreases. In the other case, function F_1 does not increase and F_2 decreases. Then the bounded function $F = (F_1, F_2)$ lexicographically ordered, decreases when any rule is applied. They also proved that in any unstable configuration at least one processor applies a rule in each computation step. Since F_2 is bounded from below, within a finite number of computation steps F will reach its minimum value, which indicates that a spanning tree is formed. Huang and Chen leaves the complexity analysis as a future study.

Another self-stabilizing algorithm for breadth-first spanning tree construction was presented by Dolev, Israeli and Moran [7, 8] for the shared memory model assuming read/write atomicity. The algorithm is semi-uniform and dynamic. The model is less restricted and more realistic than that of Huang and Chen [11]. Each processor continuously computes its distance from the root (the special processor) and notifies the result to all its neighbours. The way of computing the distance from the root is to read all its neighbours' distances, choose the minimum from among them and then add one.

In the same paper, Dolev, Israeli and Moran have proposed a nice technique, *fair combination*, we also call it *fair composition*, for designing self-stabilizing algorithms. The main idea is to compose two or more self-stabilizing algorithms, each algorithm takes a step alternatively, to obtain a more complex algorithm. Theorem 5.4 [8] states :" Assume that AL_2 is self-stabilizing for a task T_2 given task T_1. If AL_1 is self-stabilizing for T_1, then the fair combination of AL_1 and AL_2 is self-stabilizing for T_2." Dolev, Israeli and Moran composed their bread-first spanning tree algorithm with a mutual-exclusion algorithm for a tree system. The composed algorithm acts as a compiler, which makes it possible for any protocol that is self-stabilizing under composite atomicity to execute correctly in a self-stabilizing style under only read/write atomicity.

CHAPTER 3

Prepatory Graph Theory and Distributing Computing Theory

3.1 Graph Theory Preliminaries

3.1.1 Notation

For this entire thesis, let graph $G = (V, E)$ denote a connected, weighted, undirected graph where V are the vertices, and E are the edges. An *edge* is an ordered pair $(\langle v_1, v_2\rangle$, $\mathrm{wt}(\langle v_1, v_2\rangle))$ where $\langle v_1, v_2\rangle$ is an unordered pair of vertices in V, and $\mathrm{wt}(\langle v_1, v_2\rangle)$ is a positive integer called the weight of edge $\langle v_1, v_2\rangle$. A *path* from a vertex v_0 to vertex v_k in graph G is a sequence $\langle v_0, \ldots, v_k\rangle$ of vertices such that $\langle v_i, v_{i+1}\rangle \in E$ for $i = 0, \ldots, k-1$. A *cycle* is formed by a path $\langle v_0, \ldots, v_k\rangle$ if $v_0 = v_k$ and $v_1, \ldots, v_k$ are distinct. A connected graph is one in which a path exists between every pair of vertices. If $e_i = \langle v_i, v_{i+1}\rangle$ for i from 0 to $k-1$, then the path $\langle v_0, \ldots, v_k\rangle$ is also recorded as $e_0, \ldots, e_{k-1}$. Similarly, cycle $\langle v_0, \ldots, v_k\rangle$ is also recorded as the edges $e_0, \ldots, e_{k-1}$ where $e_{k-1} = \langle v_{k-1}, v_0\rangle$ and $e_i = \langle v_i, v_{i+1}\rangle$ for i from 0 to $k-2$.

A spanning tree of graph G is a subset of $\mid V \mid -1$ edges from E connecting all the vertices in G.

In the following we assume all edge weights are distinct. The size of V is n and the size of E is m. A spanning tree is minimum if the sum of the weights of its edges is as small as possible. If weights of the edges are distinct, then the minimum spanning tree is unique.

3.1.2 The Minimum Spanning Tree Problem and Properties

The minimum spanning tree (MST) problem is described as follows:

Input: A graph $G = (V, E)$

Output: A minimum spanning tree $MST(G) = (V, \hat{E})$, where $\hat{E} \in E$.

MST can be computed efficiently using algorithms that are greedy in nature. One such algorithm, known as Kruskal's algorithm [3], maintains a forest that is initially the forest containing all nodes in V and no edges. First the edges of the graph are sorted in nondecreasing order of their weights. The edges of the graph are processed in sorted order, and if the addition of the current edge does not create a cycle in the current forest, it is added to the forest. Eventually the algorithm adds $\mid V \mid -1$ edges to generate a minimum spanning tree.

In what follows, we develop a new algorithm that finds the minimum spanning tree of a graph assuming that an arbitrary spanning tree is already known. This algorithm is less efficient in the sequential setting than well-known greedy solutions to the general MST problem such as Kruskal's and Prim's algorithms [3], however, it has some properties that are appealing in concurrent settings and adapt well to self-stabilization.

Throughout $T = (V, E')$ denotes a spanning tree of $G = (V, E)$. An edge $e \in E'$ is called a *tree edge* and $e \in E \setminus E'$ is a *non-tree edge*.

If $\langle v_0, v_1 \dots v_k \rangle$ is a path in T, and $\langle v_k, v_0 \rangle$ is an edge in $E \setminus E'$, then the cycle $\langle v_0, \dots, v_k, v_0 \rangle$ in G is called a *fundamental cycle* of T containing $\langle v_k, v_0 \rangle$.

Claim 3.1.1 *$\forall e \in E \setminus E'$, there is exactly one fundamental cycle of T containing e.*

Proof: Let $e = \langle v_1, v_2 \rangle \in E \setminus E'$. Because T is a spanning tree of G, there is a path in T between any two vertices. So let P be a path in T between v_1 and v_2. Therefore edge

$\langle v_1, v_2 \rangle$ and the path P form a fundamental cycle. If such a fundamental cycle is not unique, there must be two paths between v_1 and v_2 in the tree T. Together, some edges in these two paths form a cycle and all these edges are in T, contradicting that T is a tree. Therefore a spanning tree of a connected graph and one of its non-tree edges has a unique fundamental cycle. ■

Let *fnd_cyl*(T, e) denote the unique fundamental cycle of $T = (V, E')$ containing $e \in E \setminus E'$. When the edge set E' of T is needed, it is denoted *fnd_cyl*(E', e).

Claim 3.1.2 *If e is non-tree edge and $e' \in$ fnd_cyl(E', e), then $(E' \cup \{e\}) \setminus e'$ is the edge set of a spanning tree of G*

Proof: Let $E^* = (E' \cup \{e\}) \setminus e'$. Since E' is the edge set of a spanning tree, by Claim 3.1.1 fnd_cyl(E', e) is unique. E^* is obtained from E' by adding e and removing e', so E^* still contains no cycle and still has size $\mid V \mid -1$. Therefore E^* is an edge set of a spanning tree of G. ■

Claim 3.1.3 *If e is in $MST(G)$, then e is not a maximum edge in any cycle of G.*

Proof: Assume $e = \langle v_1, v_2 \rangle \in MST(G) = (V, \hat{E})$ and e is the maximum edge of some cycle $e, e_1, e_2, \ldots, e_k$ of G. Clearly not all of $e_1, e_2, \ldots, e_k$ are also in $\hat{E}$. Suppose there is an $e_i \in E \setminus \hat{E}$, such that the fnd_cyl$(\hat{E}, e_i)$ contains e. Then the edge set $(\hat{E} \bigcup \{e_i\}) \setminus \{e\}$ forms a spanning tree of G, by Claim 3.1.2, with less total weight than $\hat{E}$, contradicting the fact that $\hat{E}$ is the edge set of the minimum spanning tree. Therefore, for every edge e_i in the cycle $e, e_1, \ldots, e_k$ that is not in $\hat{E}$, the fnd_cyl$(\hat{E}, e_i)$ does not contain e. Now construct the following path P. Initially P is empty. For each e_i, $1 \leq i \leq k$, if $e_i \in \hat{E}$, append e_i to P, if $e_i \in E \setminus \hat{E}$, append fnd_cyl$(\hat{E}, e_i) \setminus \{e_i\}$ to P. Now P is a path in $MST(G)$, from v_1 to v_2 that does not contain $e = \langle v_1, v_2 \rangle$. Hence P concatenated with e must contain a cycle in $MST(G)$, which is impossible. ■

Denote by $max(\text{fnd\_cyl}(E', e))$, the edge with maximum weight in $\text{fnd\_cyl}(E', e)$. For $e \in E \setminus E'$, the function *minimize_cyl*(E', e) returns a new set of edges and is defined by

$$\text{minimize\_cyl}(E', e) = (E' \bigcup \{e\}) \setminus \{\max(\text{fnd\_cyl}(E', e))\}.$$

Corollary 3.1.1 *For non-tree edge e, minimize_cyl(E', e) is an edge set of a spanning tree of G.*

Proof: This is a special case of Claim 3.1.2. ■

The next claim says that a spanning tree $T = (V, E')$ does not change after applying $\text{minimize\_cyl}(E', e)$, for any non-tree edge e, if and only if T is the minimum spanning tree.

Claim 3.1.4 *E' is the edge set of $MST(G)$ if and only if $\forall e \in E \setminus E', E' = \mathit{minimize\_cyl}(E', e)$.*

Proof: For sufficiency, suppose that $MST(G) = (V, \hat{E})$ where $\hat{E} = \{\hat{e}_1, \hat{e}_2, \ldots, \hat{e}_{n-1}\}$ and $w(\hat{e}_1) < w(\hat{e}_2) < \ldots < w(\hat{e}_{n-1})$. Let $E' = \{e'_1, e'_2, \ldots, e'_{n-1}\}$ denote the edges in T, where T satisfies $E' = \text{minimize\_cyl}(E', e), \forall e \in E \setminus\ E'$ and $w(e'_1) < w(e'_2) < \ldots < w(e'_{n-1})$. We prove that $e'_i = \hat{e}_i, \forall 1 \leq i \leq n-1$ by induction on i. For the basis, let e_{sm} be the smallest edge in E. By Kruskal's algorithm, $e_{sm} \in \hat{E}$, and thus $e_{sm} = \hat{e}_1$. Suppose $e'_1 \neq e_{sm}$. Then, by our choice of ordering, $e_{sm} \notin E'$. But then $\text{minimize\_cyl}(E', e_{sm})$ will return a new edge set of a new tree $T^* \neq T$. Therefore $e_{sm} = e'_1 = \hat{e}_1$. Now assume $\{\hat{e}_1, \hat{e}_2, \ldots, \hat{e}_k\} = \{e'_1, e'_2, \ldots, e'_k\}$, for $k < n-1$. By Kruskal's algorithm, $\hat{e}_{k+1}$ is the smallest edge not in $\{\hat{e}_1, \hat{e}_2, \ldots, \hat{e}_k\}$ such that $\{\hat{e}_1, \hat{e}_2, \ldots, \hat{e}_k\} \cup \{\hat{e}_{k+1}\}$ is acyclic. Therefore $w(e'_{k+1}) \geq w(\hat{e}_{k+1})$. If $e'_{k+1} \neq \hat{e}_{k+1}$, then by the choice of ordering, $\hat{e}_{k+1} \notin E'$, and the graph $(V, E' \bigcup \{\hat{e}_{k+1}\})$ will contain a cycle. Because $\{\hat{e}_1, \hat{e}_2, \ldots, \hat{e}_{k+1}\}$ is acyclic, $\{e'_1, e'_2, \ldots, e'_k\} \cup \{\hat{e}_{k+1}\}$ is acyclic. So the $\max(\text{fnd\_cyl}(E', \hat{e}_{k+1}))$ must be contained in set $E' \setminus \{e'_1, \ldots e'_k\} = \{e'_{k+1}, \ldots, e'_{n-1}\}$, say e_m. Therefore $\text{minimize\_cyl}(E', \hat{e}_{k+1})$ will replace e_m with $\hat{e}_{k+1}$. So $\text{minimize\_cyl}(E', \hat{e}_{k+1})$ $\neq E'$ which contradicts the condition of the claim. Thus, E' contains the same edges as $\hat{E}$. So $T = (V, E') = MST(G)$.

For necessity, observe that if minimize_cyl(E', e) returns an edge set E^* of a tree, say T^*, different from T, then T^* is a spanning tree with weight strictly less than T, which is impossible if T is the minimum spanning tree of G. ■

The next claim says that once minimize_cyl removes an edge from a spanning tree by replacing it with a lighter one, no subsequent application of minimize_cyl can put that edge back into the spanning tree.

Claim 3.1.5 *Consider a sequence of spanning trees $T_0 = (V, E_0), T_1 = (V, E_1), \ldots$, where T_0 is any spanning tree of G, and for $i \geq 1$, $E_i =$ minimize_cyl(E_{i-1}, e_{i-1}) for some edge $e_{i-1} \in E \setminus E_{i-1}$. Let $e_i^* =$ max(fnd_cyl(E_{i-1}, e_{i-1})). Then $\forall i \geq 1, e_i^* \notin E_j$ for any $j \geq i$.*

Proof: We use induction on j to prove, the stronger result that $\forall i \geq 1$ and $\forall j \geq i$, $e_i^* \notin E_j$ and $e_i^* = \max(\text{fnd\_cyl}(E_j, e_i^*))$.

For the basis, consider $j = i \geq 1$. Since $e_i^* = \max(\text{fnd\_cyl}(E_{i-1}, e_{i-1}))$, $e_i^* \notin E_i$ and clearly $e_i^* = \max(\text{fnd\_cyl}(E_i, e_i^*))$. Now assume the inductive hypothesis, namely that $e_i^* \notin E_{i+k}$ and $e_i^* = \max(\text{fnd\_cyl}(E_{i+k}, e_i^*))$ for $k \geq 0$. Consider $E_{i+k+1} = \text{minimize\_cyl}$ (E_{i+k}, e_{i+k}) where e_{i+k} is some edge in $E \setminus E_{i+k}$. If $E_{i+k+1} = E_{i+k}$, then obviously $e_i^* \notin E_{i+k+1}$. So suppose $e_{i+k+1}^* = \max(\text{fnd\_cyl}(E_{i+k}, e_{i+k})) \neq e_{i+k}$.

Case I: $e_{i+k+1}^* \notin \text{fnd\_cyl}(E_{i+k}, e_i^*)$, then $\text{fnd\_cyl}(E_{i+k+1}, e_i^*) = \text{fnd\_cyl}(E_{i+k},\ e_i^*)$. So $e_i^* = \max(\text{fnd\_cyl}(E_{i+k+1}, e_i^*))$ and $e_i^* \notin E_{i+k+1}$.

Case II: $e_{i+k+1}^* \in \text{fnd\_cyl}(E_{i+k}, e_i^*)$. Then T_{i+k+1} is formed from T_{i+k} by removing e_{i+k+1}^* and adding e_{i+k}. Let $\text{fnd\_cyl}(E_{i+k}, e_i^*)$ be $e_i^* P_1 e_{i+k+1}^* P_2$ and let $\text{fnd\_cyl}(E_{i+k}, e_{i+k}) = e_{i+k} P_3 e_{i+k+1}^* P_4$ where P_1, P_2, P_3 and P_4 are the paths in T_{i+k} directed so that the end points of e_{i+k+1}^* are in the same order in both lists. Let $\overline{P}$ denote the path formed from the edges of P takes in the reverse order. Then $\text{fnd\_cyl}(E_{i+k+1}, e_i^*)$ is contained in the path $e_i^* P_1 \overline{P_3} e_{i+k} \overline{P_4} P_2$. Since $\text{wt}(e_i^*) \geq \text{wt}(e), \forall e \in P_1 \bigcup P_2 \bigcup e_{i+k+1}^*$ and $\text{wt}(e_{i+k+1}^*) \geq \text{wt}(e), \forall e \in P_3 \bigcup P_4 \bigcup \{e_{i+k}\}$, $e_i^* = \max(\text{fnd\_cyl}(E_{i+k+1}, e_i^*))$ and $e_i^* \notin E_{i+k+1}$. ■

The preceding claims combine to provide the strategy for an algorithm that converts an arbitrary spanning tree into the MST. Specifically, if minimize_cyl is applied successively for each non-tree edge in any order for any initial spanning tree, the results is the minimum spanning tree.

Claim 3.1.6 *Let $T_0 = (V, E_0)$ be a spanning tree of $G = (V, E)$ and $\{e_1, e_2, \ldots, e_{m-n}\} = E \setminus E_0$ (in any order). Let $E_i = minimize\_cyl(E_{i-1}, e_i)$, for $i = 1$ to $m-n$. Then $T_{m-n} = (V, E_{m-n}) = MST(G)$.*

Proof: Claim 3.1.5 implies that $\forall e \in E \setminus E_{m-n}, e = \max(\text{fnd\_cyl}(E_{m-n}, e))$. So $E_{m-n} = \text{minimize\_cyl}(E_{m-n}, e), \forall e \in E \setminus E_{m-n}$. So by Claim 3.1.4, E_{m-n} is the edge set of $MST(G)$. ■

Claim 3.1.7 *Let $E' \subset E$ be the edges of a spanning tree of $G = (V, E)$ and $w \in E \setminus E'$. Let $y = max(fnd\_cyl(E', w))$ and $E'' = (E' \cup \{w\}) \setminus \{y\}$. $\forall e \notin E' \cup E''$, either $max(fnd\_cyl(E', e)) = max(fnd\_cyl(E'', e))$ or $max(fnd\_cyl(E', e)) = y$.*

Proof: Case I: $y \notin \text{fnd\_cyl}(E', e)$, then $\text{fnd\_cyl}(E'', e) = \text{fnd\_cyl}(E', e)$. So

$$\max(\text{fnd\_cyl}(E', e)) = \max(\text{fnd\_cyl}(E'', e)).$$

Case II: $y \in \text{fnd\_cyl}(E', e)$. Then E'' is obtained from E' by removing y and adding w. Let $\text{fnd\_cyl}(E', e)$ be eP_1yP_2 and let $\text{fnd\_cyl}(E', w) = wP_3yP_4$ where P_1, P_2, P_3 and P_4 are the paths in the spanning tree directed so that the end points of y are in the same order in both lists. Let $\overline{P}$ denote the path formed from the edges of P taken in the reverse order. Then $\text{fnd\_cyl}(E'', e)$ is contained in the path $eP_1\overline{P_3}w\overline{P_4}P_2$. Since $y = \max(wP_3yP_4)$, $y > \max(\overline{P_3}w\overline{P_4})$. If $\max(eP_1yP_2) \in \{e\} \cup \{P_1\} \cup \{P_2\}$, then $\max(eP_1yP_2) = \max(eP_1\overline{P_3}w\overline{P_4}P_2)$, so $\max(\text{fnd\_cyl}(E', e)) = \max(\text{fnd\_cyl}(E'', e))$, so the Claim holds. If $\max(eP_1yP_2) \notin \{e\} \cup \{P_1\} \cup \{P_2\}$, then $\max(eP_1yP_2) = y$, so the Claim also holds. ■

Consider a spanning tree that is changing over time by a sequence of steps, each of which adds a non-tree edge and removes the tree edge of maximum weight in the fundamental cycle of that edge. Even though the fundamental cycles of other edges are also changing by these actions, the next Corollary say that the maximum edges in this fundamental cycles do not change unless they are removed from the tree.

Let $T_0 = (V, E_0)$ is a spanning tree. Let $X = \{(e,m) \mid e \notin E_0 \text{ and } m = \max(\text{fnd\_cyl}(T_0, e))\}$.

Corollary 3.1.2 *Consider the procedure:*

$j \leftarrow 0$

repeat

choose $(e,m) \in X$

if $e \notin T_j$ *and* $m \in T_j$ *then* $T_{j+i} \leftarrow (T_i \cup \{e\}) \setminus \{m\}$

$j \leftarrow j+1$

For any j*, for any* $(e,m) \in X$*, if* $e \in E \setminus T_j$ *and* $m \in T_j$*, then* $m = max(fnd\_cyl(T_j, e))$.

Proof: This follows directly from Lemma 3.1.7 by induction. ■

3.1.3 Distributed Computing Model

An asynchronous distributed message-passing network of processors is modelled by a simple, weighted, connected and undirected graph where vertices represent processores, edges represent communication links between processores and weights represent some measure of the cost of communicating over the corresponding link.

Each processor P has a distinct identifier, and knowledge only of the identifiers of its neighbours and for each neighbour Q, the weight of the edge $\langle P, Q \rangle$. Edge weights are assumed to be distinct since they can always be made so by appending to each weight the identifiers of the edge's end-points. For example, an edge of weight w between two vertices with x and y (where $x < y$) can be encoded as the triple (w,x,y) instead, making it unique. Then weights are ordered lexicographically on these triples.

The self-stabilizing MST problem requires that given any initial configuration of the network, each processor is required to determine for each of its adjacent edges, whether or not it is in the minimum spanning tree of the network.

Self-stabilization is impossible for purely asynchronous message-passing systems [10]. We therefore assume that each processor in the network is augmented with a time-out mechanism that satisfies a necessary safety property, namely: each processor's time-out interval is guaranteed to be at least as long as the time taken by any message sent by the processor to travel a path of n edges where n is the number of processors in the network. For correctness we require that this lower bound on the time-out interval is not violated but it can be any (even very large) overestimate. The time-out interval could be provided directly or could be described as n times α, when α is the maximum time for any message to travel any edge. In the first case, knowledge of n is not required; in the second case this is the only place when knowledge of n is used. Of course, since our second algorithm is self-stabilizing, a violation in the safety of a time-out can, at worst, act as a fault from which the algorithm will eventually recover.

CHAPTER 4

From An Arbitrary Spanning Tree To A Minimum Spanning Tree

From Claim 3.1.6 we see that the application of minimize_cyl to each of the non-tree edges results in the MST regardless of the order of application. This suggests that the minimize_cyl operations could proceed concurrently provided care is taken that they do not interfere with each other. This is the central idea for a distributed algorithm, called Basic_MST which identifies the minimum spanning tree of a network provided the network has already identified an arbitrary spanning tree.

The description of algorithms Basic_MST is simplified by temporarily changing perspective to one where we pretend that communication edges do the processing and that nodes act as message-passing channels between adjacent edges. Call the graph representing this particular network setting *altered(G)*. That is, given a message-passing network of processors modeled by a graph G, we describe our algorithm for the network that is modeled by altered(G) where each edge has access to the identifiers of the edges incident at each of its end-points. It is not difficult to show how the original network, G, simulates an algorithm designed for the network altered(G). Section 6.1 contains a formal description of how this is done.

In algorithm Basic_MST, each non-tree edge e initiates a messages that searches for a heavier edge in the fundamental cycle consisting of e itself and the current tree edges. If one is found then edge e initiates another procedure that removes the heavy edge and adds e to the collection of tree edges.

Of course, non-tree edges proceed concurrently to search fundamental cycles and adjust spanning tree membership. However, we will see that Basic_MST manages these concurrent changes so that errors do not result.

Algorithm Basic_MST is described in Section 4.1, proved correct in Section 4.2 and analyzed in Section 4.3. Some properties of algorithm Basic_MST will be discussed in Sections 4.4 and 4.5.

4.1 A Distributed MST algorithm given a spanning tree

4.1.1 The Edge Processors

We assume that vertices in the network have distinct identifiers taken from some totally ordered set *ID* (with order relation denoted $<$). Let *EID* = ID $\times$ ID. An edge processor e is a member of EID. If $e = \langle u, v\rangle$, then u and v are the distinct identifiers of the two end-points of the edge processor e. Each edge processor has a *weight* that is a positive integer. The weight of edge e is return by the function $\mathrm{wt}(e)$. Recall that distinct weights of edges can be assumed.

The variables $e = \langle u, v\rangle$ and $\mathrm{wt}(e)$ are assumed to be static and in uncorruptable stable storage. Of course, the program for an edge $\langle u, v\rangle$ is also stable.

Each edge processor $e = \langle u, v\rangle$ also maintains the following three local variables in unstable storage:

- The boolean *chosen_status*, which indicates whether or not the edge processor e currently is in the Chosen_Set subgraph. We call e a *chosen edge* if chosen_status(e) is True and an *unchosen edge* otherwise.

- The non-negative integer *timer* in the interval $[0, 3 \times \mathrm{safetime}_e]$, where $\mathrm{safetime}_e$ is an upper bound on the time required for a message sent by e to travel any path of length at most n.

- The ID *end-point* $\in \{u, v\}$ that is used to record from which end-point the current message arrived.

These three values are all variables that are read and written by the algorithm Basic_MST.

4.1.2 The Messages

There are three types of messages. A *Search* message has 3 fields ("search", *eid*, *weight*), where *eid* is a member of EID and *weight* is an edge weight. A *Replace* messages has 3 fields ("replace", *weight*, *path*), where *path* is a list of EID's recording a path of chosen edges. A *Crowned* message has 2 fields ("crowned", *path*).

The purpose of Search messages is to find the heaviest weight chosen edge in the fundamental cycle that contains unchosen edge e. Replace messages and Crowned messages replace the chosen edge found by the Search message with the unchosen edge that initiated the Search message.

4.1.3 The Protocol

An edge processor is adjacent to other edge processors at each of its end points. For edge processor with $e = \langle u, v \rangle$ denote those edge processors adjacent to end-point u (respectively v) by $N(u)$ (respectively $N(v)$). We also denote the end point v of $\langle u, v \rangle$ by $\overline{u}$.

An edge $e = \langle u, v \rangle$ employs two procedures for sending messages to its neighbouring edges. The procedure $send(mess, e_{neigh})$ sends the message *mess* to the neighbouring edge processor e_{neigh}. (The algorithm ensure that e_{neigh} is a neighbour of e). The procedure $propagate(mess, \textit{end-point})$ sends a copy of message *mess* to all edge processors in $N(\textit{end-point})$, where *end-point* $\in \{u, v\}$. When a message is received, the end-point it arrived at is recorded. Typically, the propagate procedure is used to forward copies of that message (possibly revised) to neighbouring edge processors at the opposite end-point.

Define e' to be a *destination* edge of a Replace message ("replace", *weight*, *path*), if $\mathrm{wt}(e') = weight$. Define e to be a *destination* edge of a Crowned message ("crowned", *path*), if $\mathrm{last}(path) = e$.

When an unchosen edge $e = \langle u, v \rangle$ times-out, edge $\langle u, v \rangle$ propagates, through end-point v, a Search message that has been initialized to ("search", $\langle u, v \rangle$, $\mathrm{wt}(e)$) and resets its timer. The propagation continues along chosen edges but terminates at any unchosen edge. As the propagating procedure proceeds, the weight field of the Search message is updated to contain the heaviest weight of any edge covered by the Search message. When unchosen edge e receives its own Search message ("search", $\langle u, v \rangle$, *weight*), it compares it own weight, $\mathrm{wt}(e)$, with *weight* recorded in the Search message. If $weight = \mathrm{wt}(e)$ then e becomes passive until the next time-out. Otherwise e propagates a Replace message initialized to ("replace", *weight*, $[e]$) [1]. When a Replace message is received by a chosen edge $\hat{e}$ whose weight $\mathrm{wt}(\hat{e})$ is not equal to *weight*, then $\hat{e}$ prepends its own EID to the head of the path and propagates the modified Replace message again. Just as for Search messages, Replace messages terminate at unchosen edges. When a Replace message ("replace", *weight*, *path*) is received by its destination edge e', then e' removes itself from the current Chosen_Set by setting its chosen_status to False and initiates the Crowned message ("crowned", *path*). The Crowned message follows the path of edges constructed by the Replace message in reverse order, back to the unchosen edge e. When the Crowned message is received by its destination edge e, e puts itself into the Chosen_Set by setting its chosen_status to True.

The Basic_MST algorithm is shown in Figure 4.1.3. It uses two functions. The function *reset_timer* is a built-in function for each edge processor, e, which initializes the timer to 2 $\times$ safetime(e). For $list = [x_1, x_2, \ldots, x_n]$, $head(list) = x_1$ and $tail(list) = [x_2, \ldots, x_n]$. The symbol λ denotes the empty list. The symbol $\oplus$ is used for concatenation of the list.

[1] List are delimited with square brackets i.e. $[\ldots]$

Procedure for edge processor $e = \langle u, v \rangle$:

`Upon time-out`
if $\left(\neg \text{ chosen\_status}(e)\right)$ then
  `propagate` $\left((\text{“search”}, \langle u, v \rangle, \text{wt}(e)), \text{v}\right)$;
  `reset_timer`;

`Upon receipt of` (“search”, *eid*, *weight*) `from` $\textit{end-point} \in \{u, v\}$
  if $\left(\text{chosen\_status}(e)\right) \wedge \left(e \neq \textit{eid}\right)$ then
    `propagate` $\left(\left(\text{“search”}, \textit{eid}, \max(\text{wt}(e), \textit{weight})\right), \overline{\textit{end-point}}\right)$;
  elseif $\left(e = \textit{eid}\right) \wedge \left(\neg \text{ chosen\_status}(e)\right) \wedge \left(\textit{weight} > \text{wt}(e)\right)$ then
    `propagate` $\left((\text{“replace”}, \textit{weight}, [\text{e}]), \overline{\textit{end-point}}\right)$;
    `reset_timer`;

`Upon receipt of` (“replace”, *weight*, *path*) `from` $\textit{end-point} \in \{u, v\}$
  if $\left(\text{chosen\_status}(e)\right) \wedge \left(\text{wt}(e) \neq \textit{weight}\right)$ then
    `propagate` $\left((\text{“replace”}, \textit{weight}, [e] \oplus \textit{path}), \overline{\textit{end-point}}\right)$;
  elseif $\left(\text{chosen\_status}(e)\right) \wedge \left(\text{wt}(e) = \textit{weight}\right)$ then
    chosen_status(e)$\leftarrow$ False;
    `send`$\left((\text{“crowned”}, \textit{path}), \text{head}(\textit{path})\right)$;
    `reset_timer`;

`Upon receipt of` (“crowned”, *path*)
  if $\left(\text{tail}(\textit{path}) \neq \lambda\right)$ then
    `send`$\left((\text{“crowned”}, \text{tail}(\textit{path})), \text{head}(\text{tail}(\textit{path}))\right)$;
  elseif $\left(\text{tail}(\textit{path}) = \lambda\right)$ then
    chosen_status(e) $\leftarrow$ True;

Figure 4.1: Algorithm Basic_MST

4.2 Proof of Correctness

We prove that if algorithm Basic_MST is run from any initial configuration where the edges with chosen_status = True form a spanning tree of the network and there are no messages in the network, then eventually, those edges with chosen_status = True will be exactly the edges of the minimum spanning tree and will subsequently not change. Consider any execution of algorithm Basic_MST beginning from any such initial configuration where there are no messages in the system and the edge processors with chosen_status = True constitutes a spanning tree of the network. The execution proceeds in steps that are determined by a weakly fair scheduler. At step i, the processor chosen by the scheduler executes its next atomic action.

Several definitions will help make the proof precise and concise.

Define *local_state(e, i)* to be the sequence of values of chosen_status(e), timer(e) and the collection of messages at e at step i. At any step i, we capture the important attributes of the state of the entire system by the *configuration at step i*, denoted *Config(i)*, and defined by: Config(i) = (local_state(e_1, i), local_state(e_2, i), $\ldots$, local_state(e_m, i)). Define *Chosen_Set(i)* to be the set $\{e \mid \text{In Config}(i), \text{chosen\_status}(e) = \text{True}\}$.

The proof depends on capturing each edge that, while not yet chosen, has a Crowned message destined to it that will change its chosen_status to True. We call this the *latent_status*, which is not a variable of the algorithm, but is only used for the proof, and is defined as follows. At step 0, for every edge e, latent_status(e) is False. Latent_status(e) becomes True at step i if, at step i, a chosen edge e' changes its chosen_status to False because of receipt of a Replace message (“`replace`”, wt(e'), *path*) where head(*path*) $= e$. Latent_status(e) becomes False at step j if at step j edge e change its chosen_status to True because of receipt of a Crowned message (“`crowned`”, *path*), where head(*path*) $= e$. Define Latent_Set(i) to be the set $\{e \mid \text{In Config}(i), \text{latent\_status}(e) = \text{True}\}$. Define *Target_Set(i)* to be the set Chosen_Set(i) $\cup$ Latent_Set(i).

Define *major steps* to be the subsequence of steps at which some processor executes either line 12 and 18 of algorithm Basic_MST. Denote the subsequence of $(1, 2, 3, \ldots)$ of these steps that are major steps by $(t_1, t_2, \dots)$. Observe that Chosen_Set can only change at line 12 or 18. Also, by definition, Latent_Set can only change when Chosen_Set does. So Target_Set can only change at major steps.

As a Search message propagates, many copies are produced that travel over the network. We use *Search_Message_Set(s,e)* to denote all of the copies of a Search message s that was initiated by an unchosen edge e. If on the network, one search and replace procedure completed before another began, then for each unchosen edge, e, the fundamental cycle of e would remained fixed during the search procedure and each unchosen edge e would receive exactly one copy from Search_Message_Set(s,e). All the other copies would terminate at leaves of the current tree. Because of the concurrency and asynchrony, however, fundamental cycles are changing dynamically and it is possible that an unchosen edge, e, receives multiple copies from Search_Message_Set(s,e) . This can happen if some edge $\tilde{e}$ in the fundamental cycle of e is the maximum weight edge in the fundamental cycle of some other unchosen edge $\hat{e}$. Then it is the possible that one copy from Search_Message_Set(s,e) travels e's original fundamental cycle and then $\hat{e}$ removes $\tilde{e}$ and adds $\hat{e}$ to Chosen_Set, thus changing the fundamental cycle of e. Then another copy from Search_Message_Set(s,e) (moving slowly) travels this revised fundamental cycle. Therefore it is possible that one initial Search message, s (initiated in line 2) generates several replace responses (line 7). A Replace message is *associated* with an initial_search_message, s, if it is generated in response to some message $\hat{s}$ in Search_Message_Set(s,e_s). A Replace message is *successful* if, when it is received by its destination edge, e_r, e_r is chosen. A successful Search message causes the receiving edge to become unchosen and a Crowned message to be sent to the sender of the search thus adding it to the Latent_Set.

Our goal is to show that the Target_Set is always a spanning tree. The proof of this hinges on the property that there is at most one successful Replace message associated with any Search_Message_Set(s,e). The details are in the following lemma, which depends on some graph theoretic lemmas from Chapter 3.

Lemma 4.2.1 *For every step i, Target_Set(i) is a spanning tree.*

Proof: Since Target_Set can only change when at least one of Chosen_Set or Latent_Set changes, it suffices to prove the lemma for major steps. Let t_0 be step 0 and t_i be the ith major step, $i \geq 1$. The proof proceeds by induction on the major step index. To achieve the proof, we strengthen the induction hypothesis to:

1. For every $i \geq 0$, Target_Set(t_i) is a spanning tree and
2. Target_Set$(t_i) = ($Target_Set$(t_{i-1}) \cup \{e\}) \setminus \{\max(fnd_cyl($Target_Set$(t_{i-1}), e))\}$ for some e such that chosen_status$(e) =$ False at step $t_i - 1$.

Initially the Chosen_Set(t_0) is a spanning tree and there are no messages in the system. So trivially Latent_Set$(t_0) = \emptyset$, and thus Target_Set$(t_0) =$ Chosen_Set$(t_0) \cup$ Latent_Set(t_0) is a spanning tree. When Target_Set(t_0) changes to Target_Set(t_1), it is the first time that a Replace message ever succeeds. All the unchosen edges at step t_0 remains False at step t_1, including e which initiated the Replace message and no fundamental cycles have yet changed. However, at time t_1, edge e is added to the Latent_Set. Therefor Target_Set$(t_1) =$ $($Target_Set$(t_0) \cup \{e\}) \setminus \max(fnd_cyl($Target_Set$(T_0, e)))$ and chosen_status$(e) =$ False at step $t_1 - 1$. By Corollary 3.1.1 Target_Set(t_1) is a spanning tree. Therefore the inductive hypothesis holds for the base cases t_0 and t_1.

Assume that for every $0 \leq j < k$, Target_Set(t_j) is a spanning tree and Target_Set$(t_j) =$ $($Target_Set$(t_{j-1}) \cup \{e\}) \setminus \{\max(fnd_cyl($Target_Set$(t_{j-1}), e)\}$ for some edge e, where chosen_status$(e) =$ False at the step $t_j - 1$.

Case I. Suppose line 18 is executed by some edge e_s at a major step t_k. Then the most recent message received by this edge e_s was a Crowned message with destination e_s. Thus at step $t_k - 1$, latent_status$(e_s) =$ True, and at step t_k chosen_status$(e_s) =$ True. So Target_Set remains unchanged. That is Target_Set$(t_k) =$ Target_Set(t_{k-1}). So the inductive hypothesis holds in this case.

Case II. In step t_k, line 12 is executed by some edge e_r. Then the most recent message received by e_r was a Replace message with destination e_r from some edge e_s and

$$\begin{aligned} \text{chosen\_status}(e_r) &= \text{True at step } t_k - 1 \text{ (successful replace)} \\ \text{chosen\_status}(e_r) &= \text{False at step } t_k \\ \text{Latent\_Set}(t_k) &= \text{Latent\_Set}(t_k - 1) \cup \{e_s\} \end{aligned}$$

Thus if

$$e_s \notin \text{Target\_Set}(t_{k-1}) \tag{4.1}$$

and

$$e_r = \max(\text{fnd\_cyl}(\text{Target\_Set}(t_{k-1}), e_s)) \tag{4.2}$$

then Target_Set(t_k) is a spanning tree by Corollary 3.1.1 of Chapter 3, and this spanning tree is formed from the spanning tree at step t_{k-1} in the manner claimed by the inductive hypothesis.

So it remains to show equations 4.1 and 4.2 for this case. Suppose that $e_s \in$ Target_Set (t_{k-1}). Then there must have been an earlier Crowned message with destination e_s sent from some different edge, say e_w. Thus e_s must have sent at least 2 successful Replace messages, one to e_r and one to e_w. So both e_r and e_w were discovered to be the maximum in some cycle of chosen edges travelled by e_s's Search messages. First observe that both these Search messages were in the same search message set, say Search_Message_Set(s, e_s). This is because any successful replace will cause a Crowned message to reach the originator before the originator can time-out and initiate another search. Since by the induction hypothesis, Target_Set(t_j) was a spanning tree for j from 0 to $k-1$, these cycles were both fundamental cycles of e_s. So the fundamental cycle of e_s must have changed during the propagation of the Search message s. Without loss of generality, assume the fundamental cycle of e_s that contained e_w preceded in time the fundamental cycle that contained e_r. By the induction hypothesis, the spanning tree formed by the Target_Set was transformed from the one containing e_w to the one containing e_r by a sequence of steps each of which removed the maximum edge from some fundamental cycle and replaced it with the non-tree edge of that cycle. Find the index i such that in the sequence Target_Set(0), Target_Set$(t_1), \ldots,$ Target_Set(t_{k-1}), $e_w \in$ Target_Set(t_i) and $e_w \notin$ Target_Set(t_{i+1}). By the induction hypothesis, Target_Set(t_i) changed to t_{i+1} by the removal of e_w, and the addition of some unchosen edge e_u, where $e_w = \max(\text{fnd\_cyl}(\text{Target\_Set}(t_i), e_u))$. Thus e_w must have received a successful Replace message from e_u at which point e_w became unchosen. But e_w was also unchosen when it received the Replace message from e_s. This is impossible since a chosen edge can become unchosen only once by Claim 3.1.5. So there could not have been a previous successful Replace and hence $e_s \notin$ Target_Set(t_{k-1}).

By Corollary 3.1.2, as long as edge e_r remains in the spanning tree, it continues to be the maximum edge of the fundamental cycle of e_s. Since, e_r is chosen at step $t_k - 1$, and the chosen set is a spanning tree, it must be the maximum edge in the fundamental cycle of e_s in the spanning tree Target_Set$(t_{k-1}$. ■

From the proof of Lemma 4.2.1, it follows that Target_Set actually changes only when line 12 is executed, at which point, an edge is removed from Chosen_Set and another added to Latent_Set. Define $T_1, T_2, \ldots$ to be the subsequence of steps (and of $t_1, t_2, \ldots$) when line 12 is executed and call these steps *change steps*.

Let $W(S) = \sum_{e \in S} \text{wt}(e)$.

Lemma 4.2.2 *For every change step T_i , $W(\mathit{Target\_Set}(T_i)) > W(\mathit{Target\_Set}(T_{i+1}))$.*

Proof: According the proof of Lemma 4.2.1, at line 12, Target_Set$(T_{k+1}) = ($Target_Set$(T_k) \cup \{e\}) \setminus \{e'\}$ and $e' = \max(\text{fnd\_cyl}(\text{Target\_Set}(T_k), e))$. So $\text{wt}(e') > \text{wt}(e)$. Thus $W(\text{Target\_Set}(T_{k+1})) = W(\text{Target\_Set}(T_k)) + \text{wt}(e) - \text{wt}(e') < W(\text{Target\_Set}(T_k))$. ■

Lemma 4.2.3 *If Target_Set$(T_i) \neq MST$, then change step T_{i+1} will eventually occur.*

Proof: If Target_Set$(T_i) \neq MST$, then there exists $e \in MST$; but $e \notin$ Target_Set(T_i). So, e is an unchosen edge. The time-out mechanism will make sure the unchosen edge e initiates a Search message. Since Target_Set is a spanning tree for all the steps and all the edges in Latent_Set(T_i) become chosen edges because Crowned message always reach their destination, eventually, the Search message generated by e will return to e along e's fundamental cycle. Since $e \in MST$, e is not the maximum edge in any cycle; therefore, a Replace message will be generated by e to replace a chosen edge e'. When e' receives the Replace message, if T_{i+1} has not occurred by this time, e' is still unchosen. Then line 12 will get executed: e' will be removed from Chosen_Set and e will be added into Latent_Set. This step of execution of line 12 is T_{i+1}. So T_{i+1} will eventually occur. ■

Theorem 4.2.1 *Basic_MST ensures that there is a step I, such that* $\forall \hat{I} > I$, *Chosen_Set*$(\hat{I}) = MST$.

Proof: By Lemma 4.2.2 and Lemma 4.2.3, we conclude that eventually there is a step T_i, such that Target_Set$(T_i) = MST$. Within the next M steps, where M is the maximum safetime for all the edges in the network, all the edges in Latent_Set(T_i) become chosen edges because Crowned message always reach their destination before the destination times-out. So $\exists T_i \leq I < (T_i + M)$, Chosen_Set$(T)$ = Target_Set$(T) = MST$. Since Chosen_Set is the edge set of a minimum spanning tree, by Lemma 4.2.2, nothing changes subsequently. So $\forall \hat{I} > I$, Chosen_Set$(\hat{I}) = MST$. ■

4.3 Complexity

Since algorithm Basic_MST is deterministic, its time complexity on a network G is defined to be the maximum time over all executions and over all valid initial configuration for the system to converge to a legitimate configuration in this case a valid initial configuration is any spanning tree and no messages in the system. The legitimate configuration requires chosen_status(e) = True if and only if $e \in MST(G)$.

Theorem 4.3.1 *The time complexity for algorithm Basic_MST is* $O((m-n)M)$ *where* M *is the maximum safetime for all the edges in the network* G *with* n *nodes and* m *edges.*

Proof: In algorithm Basic_MST, within $2M$ time, at least one unchosen edge e that is in the minimum spanning tree of the network has a successful replace and after an additional M that replace has moved the latent edge to the Chosen_Set. By Lemma 4.2.2, every successful replace reduces the wt(Target_Set). Thus in at most $m-n$ repetitions of this argument, wt(Target_Set) will reach the minimum weight. The maximum time taken over all the executions is $3(m-n)M$, so the upper bound complexity for algorithm Basic_MST for taking the system to a legitimate configuration(MST) is $O((m-n)M)$. ■

The worst case time complexity bound of Lemma 4.3.1 can be realized for some initial spanning tree on some networks under a malicious scheduler. Consider the network in Figure 4.2. The network is the graph $G=(V,E)$. V has n nodes and $V=A\cup B$ where $\mid A\mid=\mid B\mid=n/2$. $A=\{(a,1),(a,2),\ldots,(a,n/2)\}$ and $B=\{(b,1),(b,2),\ldots,(b,n/2)\}$. E has $m=n^2/4+n-2$ edges and $E=E_a\cup E_b\cup E_{ab}$ where $E_a=\{\langle(a,i),(a,i+1)\rangle \mid 1\le i<n/2\}$ and $E_b=\{\langle(b,i),(b,i+1)\rangle \mid 1\le i<n/2\}$ and $E_{ab}=\{\langle(a,i),(b,j)\rangle \mid 1\le i\le n/2$ and $1\le j\le n/2\}$. Let $\forall e\in E_a\cup E_b, \mathrm{wt}(e)<n$ and $\mathrm{wt}(\langle(a,i),(b,j)\rangle)=ni+j$. Thus the edges of the MST of this network is $E_a\cup E_b\cup\{\langle(a,1),(b,1)\rangle\}$ The initial configuration is chosen such that all the edges in $E_a\cup E_b$ are chosen and all the edges in E_{ab} are unchosen except edge $\langle(a,n/2),(b,n/2)\rangle$. This is clearly spanning tree. In fact it differs from the MST by only one edge. The worse case complexity is structured as follows: All the unchosen edges times-out, they all find $\langle(a,n/2),(b,n/2)\rangle$ is the largest edge in their fundamental cycle and send their Replace message. The Replace message of edge $\langle(a,n/2),(b,n/2-1)\rangle$ is successful. In next repetition, $\langle(a,n/2),(b,n/2-2)\rangle$'s Replace message is successful, and so on. In round i, The Replace message of edge $\langle(a,n/2-i\ \mathrm{div}\ (n/2)),(b,n/2-i \bmod (n/2))\rangle$ is successful. Finally, in round $m-n+1$, $\langle(a,1),(b,1)\rangle$'s Replace message is successful. At this point, a minimum spanning tree is formed by the chosen edges. The time it takes to converge is at most $(m-n+1)M$

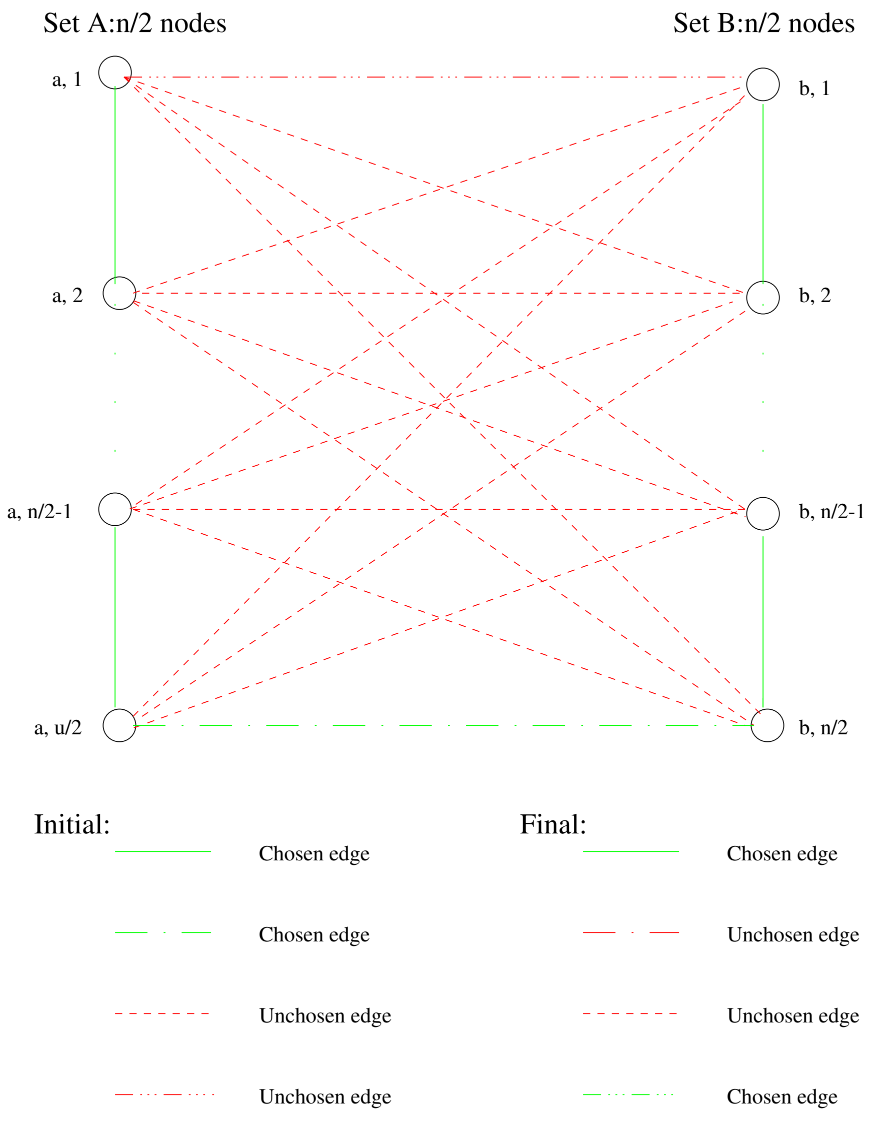

Figure 4.2: A worst case for Basic_MST

4.4 Dynamic Growing Network

Algorithm Basic_MST can be used to maintain a minimum spanning tree in a network that has edges and nodes added and weights changed dynamically.

Suppose a node is added to the network along with a set S of edges (at least one) added to link the node into the network. Arbitrarily choose one edge $e \in S$ to be chosen and for all $e' \in S \setminus \{e\}$, assign their chosen_status to False. If the original chosen set formed a spanning tree of the network, then the resulting chosen set must be a spanning tree of the new network. Therefore algorithm Basic_MST is still able to construct the a minimum spanning tree when the network dynamically gains nodes.

If an edge e is dynamically added into the network and chosen_status(e) is False, then the chosen set remains the same. If the chosen set formed a spanning tree before the change and nodes remain the same, then the chosen set still forms a spanning tree of the network. Therefore algorithm Basic_MST is still able to construct minimum spanning tree when the network dynamically gains edges.

No matter how the weights of edges change dynamically, it only effects the structure of the minimum spanning tree. The chosen set is still a spanning tree of the network. Therefore algorithm Basic_MST still works.

4.5 Bounded Message Queue

Algorithm Basic_MST works on a model with unbounded message queues. It is generally not realistic to assume a message-passing system has unbounded queues. With a little modification, the algorithm can correctly work on a message-passing network with bounded queues. Loss of a Search or a Replace message can only cause the sender of the search to time-out. These losses cannot effect the Target_Set. So throwing out the Search messages and Replace messages when the bounded queues is full will not effect the correctness of the algorithm Basic_MST. But if a Crowned message does not get to its destination, then Target_Set will no longer have the property of being a edge set of a spanning tree. So algorithm Basic_MST will not run correctly. However, each Crowned message has a path to follow to its destination. so Crowned messages do not spread all over network like the Search and Replace messages. So there is no proliferation of unnecessary Crowned messages. Therefore we can reserve the queue space for storing the Crowned messages. In order to not keep dropping the Search messages and the Replace messages from the same unchosen edge all the time, we can modify algorithm Basic_MST with different timer values. This argument also shows that Basic_MST has some properties of fault-tolerance. It can tolerant losing Search messages and Replace messages, but not Crowned messages.

CHAPTER 5

A Self-Stabilizing Minimum Spanning Tree Algorithm

This chapter presents a algorithm Self_Stabilizing_MST for the minimum spanning tree problem. Since the configuration after some faults may be arbitrary, it is more realistic and useful to design a self-stabilizing algorithm to construct a minimum spanning tree from any initial configuration. Algorithm Basic_MST guarantees convergence to the MST only if, in the initial configuration, the chosen edges form a spanning tree, and there are no messages in the network. Algorithm Self_Stabilizing_MST needs to alter and enhance Basic_MST so that the minimum spanning tree is constructed even when, initially, the chosen edges are disconnected or do not span the network or contain cycles and when there may be spurious messages already in the system.

The algorithm Basic_MST serves to provide some of the ideas for the second algorithm in a simpler but less general setting. Given Basic_MST, another approach to finding a general self-stabilizing MST algorithm might be to use the technique of fair composition [8, 6] applied to a self-stabilizing algorithm for spanning tree construction and Basic_MST. However, we failed to see how to achieve this because of the need keep the variables manipulated by Basic_MST entirely disjoint from those used to construct the spanning tree. So Self_Stabilizing_MST uses some of the ideas of Basic_MST but is developed from scratch.

The algorithm is again described for the altered graph where edges are assumed to do the processing. We use most of the notation from Chapter 4, such as Chosen_Set. The main idea of the two algorithms remains the same. In both algorithms, computation is driven by unchosen edges. However, Chosen_Set is maintained differently in the two algorithms. In algorithm Basic_MST, we momentary disconnect Chosen_Set by first removing an edge and then adding a latent edge. In Basic_MST Chosen_Set never forms a cycle during the computation. In algorithm Self_Stabilizing_MST, a new chosen edge is first added and then a latent edge is removed. The proof of correctness of algorithm Self_Stabilizing_MST will require the graph built from Chosen_Set to be connected and span the network after a certain initial period. Because we handle processing differently, this may create cycles formed by chosen edges. Therefore we also introduce we have a procedure to handle cycles.

Algorithm Self_Stabilizing_MST is described in Section 5.1, proved correct in Section 5.2, analyzed in Section 5.3.

5.1 Self-Stabilizing MST algorithm

5.1.1 The Processors

An edge processor e has an edge identifier $\langle u,v\rangle$, where u and v are the distinct identifiers of its two end-points. Let EID denote the set of edge identifiers. Each edge processor e has a *weight* that is a positive integer, and is denoted by *wt(e)*.

The identifiers of the neighbouring edges of e at its u and v end-points are in stable storage and available to e as $N(u)$ and $N(v)$ respectively.

The edge processor *description* $(\langle u,v\rangle, w)$ is assumed to be static and in uncorruptable stable storage. This description contains only local information that is determined solely by the network structure. Of course, the program for an edge $\langle u,v\rangle$ is also stable.

Each edge processor, $e = \langle u,v\rangle$, maintains three variables in unstable storage:

- A boolean *chosen_status*, which indicates whether or not the edge processor e currently is in the *Chosen_Set* subgraph.

- A non-negative integer *timer* in the interval $[0, 3 \times \text{safetime}(e)]$. Recall that $\text{safetime}(e)$ is an upper bound on the time required for a message sent by e to travel any path of length at most n.

- A boolean *search_sent*, which indicates whether edge processor e has sent a Search message that has not yet returned to e. On receiving its own Search message, search_sent will be reset to False.

These three values are all variables that are read and written by Self_Stabilizing_MST algorithm.

5.1.2 The Messages

There are three types of messages. *Search* messages have 3 fields ("search", *eid*, *path*), where *eid* is a member of EID and *path* is a list of EID's. The second field records the unchosen edge that initiates the search, and the third field records the path of chosen edges travelled by the Search message. *Remove* messages ("remove", *path*) and *Find_cycle* messages ("find_cycle", *path*), each have two fields with the second field recording a path of chosen edges.

The purpose of a Search message is to find heavy edges that should be removed from Chosen_Set. The purpose of a Remove message is to remove a heavy edge in a cycle of Chosen_Set or in a fundamental cycle of Chosen_Set. The purpose of a Find_cycle message is to detect cycles of chosen edges when there are no unchosen edges in the network.

5.1.3 The Protocol

Algorithm Self_Stabilizing_MST employs two procedures for edge $e\langle u, v\rangle$ to send a message. The procedure *send(mess, e_{neigh})* sends the message *mess* to the neighbouring edge processor e_{neigh}. (The send aborts if e_{neigh} is not a neighbouring edge of $\langle u, v\rangle$). Recall those edge processors adjacent to end-point v are denoted by $N(v)$. The procedure $propagate(mess, v)$ sends a copy of the message *mess* to all edge processors in $N(v)$.

When an unchosen edge e that is not waiting for the return of its previous Search message times-out, it initiates a new Search message ("search", e, *path*), where *path* is initially empty. When chosen edge e' receives a Search message, it first checks if there is a cycle of chosen edges in $path \cup \{e'\}$. In this case, e' initiates a Remove message with destination equal to the heaviest edge in that cycle. If there is no cycle, e' appends its eid to path and propagates the Search message through its outgoing end-point. If a Search message is received by the edge that originated it (the sender) and that edge is still unchosen, then it checks whether it is larger than all the edges in *path*. If it is the largest, then e resets its timer, sets search_sent to False and becomes passive until its next time-out. Otherwise, e changes its chosen_status to True, search_sent to False and sends a Remove message ("remove", *path*). An edge receiving a Remove message forwards it along the path recorded in the message until the Remove message reaches its destination edge. The destination edge simply sets its chosen_status to False. If e times-out before it receives its own Search message, then its two end-points are not connected by chosen edges. In this case, to guarantee that the chosen edges form a spanning connected graph, the unchosen edge adds itself to the Chosen_Set.

Algorithm Self_Stabilizing_MST also must deal with the special case when all edges are initially chosen and hence no Search messages are generated. Any chosen edge e initiates and propagates a Find_cycle message ("find_cycle", [e]) when it times-out. Like Search messages, Find_cycle messages record the list of chosen edges travelled, so a chosen edge receiving a Find_cycle message can detect if it is in a cycle of chosen edges. Then a Remove message is initiated to travel the cycle to the edge with maximum weight in that cycle, and causes that edge to set its chosen_status to False.

The intuition for the enhancements is as follows:

- Suppose the chosen edges are disconnected or do not span the network (or both).

Then there is at least one unchosen edge e whose end-points, say u and v, are not connected by a path of chosen edges. So a Search message initiated by e out of its u end-point cannot return to e's v end-point. This is detected by e through its time-out mechanism and the boolean variable search_sent, which indicates that e is waiting for the return of its Search message. When this detection occurs e simply changes its status to chosen.

- Suppose a collection of chosen edges form a cycle. To detect cycles of chosen edges, each Search message is augmented to record the list of edges on the path it travelled. If a chosen edge receives a Search message at one end-point, and the list in that Search message contains a chosen edge that is a neighbour of its other end-point, then the Search message travelled a cycle of chosen edges. This cycle-detection will succeed as long as there is an unchosen edge to initiate a Search message that will travel that cycle.

- A Search message cannot return to its initiator at the same end-point from which it was propagated. For this to happen, the Search message would have to arrive from an adjacent chosen edge. but such an edge would not propagate the Search message since it would have already detected a cycle.

- Search messages are used to establish a spanning tree as well as to convert a spanning tree into one with minimum weight.

- A Find_cycle is initiated by a chosen edge that timed-out because it did not received any Search message within an interval equal to its time-out interval. In order to avoid initiating Find_cycle messages prematurely (when there is a Search message on its way to this chosen edge) the time-out interval for any chosen edge is set to three times its safetime.

The proceed-code for algorithm Self_Stabilizing_MST is shown as follows. The functions *reset_timer* causes an edge processor with chosen_status False to reset its timer to its safetime and one with chosen_status True to reset its timer to 3 times its safetime. Timers continue to decrement and cause a time-out when they reach zero. For $list = (x_1, x_2, \ldots, x_n)$, $head(list) = x_1$ and $tail(list) = (x_2, \ldots, x_n)$. The function *max(list)* returns the maximum weight over all edges in the list. The symbol $\oplus$ denotes concatenation. Comments are delimited by brace brackets ({,}).

Procedure for edge processore = $\langle \mathbf{u}, \mathbf{v} \rangle$:

`Upon time-out:`

If (¬chosen_status) $\wedge$ (¬*search_sent*) Then
    `propagate`((“search”, $\langle u, v \rangle$, $\emptyset$), v);
    search_sent $\leftarrow$ True;
Elseif (¬chosen_status) $\wedge$ (*search_sent*) Then {disconnected chosen edges}
    chosen_status $\leftarrow$ True;
Elseif (chosen_status) Then {no searches happening}
    `propagate` $\left((\text{“find\_cycle”}, [\langle u, v \rangle]), v \right)$;
`reset_timer`.

`Upon receipt of` (“search”, *sender,* path) `from` *end-point*, `say` *u*

If (chosen_status) $\wedge$ ($sender \neq \langle u, v \rangle$) Then
    `reset_timer`;
    If ($\forall \langle v, z \rangle \in N(v), \langle v, z \rangle \notin path$) Then {no cycle}
        `propagate` ((“search”, *sender*, $path \oplus [\langle u, v \rangle]$), v);
    Else {$\exists \langle v, z \rangle \in N(v)$ s.t. $\langle v, z \rangle \in path$, so cycle of chosen}
        Let $path = list_1 \oplus list_2$ where $\texttt{head}(list_2) = \langle v, z \rangle$
        `send` $\left((\text{“remove”}, list_2), \langle v, z \rangle \right)$;
Elseif (¬chosen_status) $\wedge$ ($sender = \langle u, v \rangle$) Then
{search traversed a foundamental cycle }
    `reset_timer`;
    search_sent $\leftarrow$ False;

```
    If ( max(path) > wt(e) ) Then
        chosen_status ← True;
        send( ("remove", path), head(path) ).

Upon receipt of ("remove", path)
reset_timer;
If path is simple Then
    If ( wt(e) ≠ max(path)) Then                              {not heaviest edge}
        send( ("remove", tail(path)), head(tail(path)) );
    Else
        chosen_status ← False;
        search_sent ← False.

Upon receipt of ("find_cycle", path) from end-point, say u
reset_timer;
If (chosen_status) Then
    If (∀⟨v,z⟩ ∈ N(v), ⟨v,z⟩ ∉ path) Then
        propagate( ("find_cycle", path ⊕ [⟨u,v⟩]), v );
    Else                                        {∃⟨v,z⟩ ∈ N(v), ⟨v,z⟩ ∈ path}
        Let path = list1 ⊕ list2 where head(list2) = ⟨v,z⟩
        send ( ("remove", list2), ⟨v,z⟩ ).
```

5.2 Proof of Correctness

We prove that if algorithm Self_Stabilizing_MST is executed from any initial configuration, then eventually, those edges with chosen_status $= True$ will be exactly the edges of the minimum spanning tree and will subsequently not change. The proof technique relies on an integer valued potential function that provides a measure of how far the current Chosen_Set is from the edge set of the minimum spanning tree. We show that the value of this function is initially bounded and that it never increases and eventually decreases. Thus it must eventually equal 0 indicating that Chosen_Set is equal to MST. Let $(e_1, \ldots, e_m)$ be the edges of the network in order of increasing weight. The potential function selects the minimum index k such that for every e_i in $(e_k, \ldots, e_m)$, the chosen_status of e_i is True if and only if e_i in the edge set of MST. So when $k = 0$, the Chosen_Set is the MST.

Consider any execution of algorithm Self_Stabilizing_MST that proceeds in steps that are determined by a weakly fair scheduler. At step i, the processor chosen by the scheduler executes its next atomic action.

The definition of Config(i) and Chosen_Set(i) are as defined in Chapter 4.

We consider the behaviour of the network after the initial, spurious messages have been "worked out" of the system. Call a Search message ("search", sender, *path*), *genuine* if 1) it was initiated by an unchosen edge with edge identifier equal to sender, and 2) *path* is a non-cyclic path of edges starting at sender. Call a Find_cycle message ("find_cycle", *path*) *genuine* if it was initiated by a chosen edge e, and *path* is a non-cyclic path of edges starting at e. A Remove message is *genuine* if it was generated in response to a genuine Search message or a genuine Find_cycle message. Let M be the maximum safetime for any edge in the system. Define step S_i to be the first step that occurs after time $M \times i$.

Lemma 5.2.1 *By step S_1 all messages are genuine.*

Proof: By the definition of safetime, any Search, Find_cycle or Remove message that survives for M time must have travelled a path of length more than n. However Search and Find_cycle message stop when a cycle or a fundamental cycle is detected, which must happen before n edges are traversed. A Remove message is discarded if its path contains repeated nodes. Otherwise it can travel at most along the edges in path, of which there are at most n. ■

Lemma 5.2.2 *Let $\hat{V} \subseteq V$ such that the subgraph of $(V, Chosen\_Set(i))$ induced by $\hat{V}$ is connected and $i \geq S_1$. Then $\forall$ step $j > i$, the subgraph of $(V, Chosen\_Set(j))$ induced by $\hat{V}$ is connected.*

Proof: Algorithm Self_Stabilizing_MST removes an edge from Chosen_Set(i) only upon receipt of a Remove message (line 25). Since $i \geq S_1$ such a message is genuine, and hence was created when a cycle was discovered. This message can remove only the unique edge with maximum weight in that cycle. So at most one edge can be removed from the cycle. Thus $\hat{V}$ must remain connected by chosen edges. ■

Lemma 5.2.3 *For any initial $Config(0)$, $\forall$ steps $i \geq S_3$, the subgraph $(V, Chosen\_Set(i))$ is connected and spans the network.*

Proof: Assume Chosen_Set(i) is disconnected or does not span G for all $i, S_1 \leq i \leq S_3$. Let (V_1, V_2) be any two subsets of V such that the edges on paths between V_1 and V_2 are unchosen for the interval from S_1 to S_3. By step S_2, each of these unchosen edges will have timed-out, sent a Search message and set search_sent to True. By step at most S_3 none will have had its Search message that it initiated returned to its opposite end-point, so each will have timed-out again. Because search_sent is True each will change its chosen_status to True. By Claim 5.2.2, once V_1 and V_2 are connected they remain connected. ■

It is easy to check that if the network is itself a tree, then by step S_2 all edges are chosen and will remain so. Thus Self_Stabilizing_MST is correct of any tree network. The remainder of this proof assumes that the network G is not a tree.

Several definitions will help make the proof of the next claim precise and concise.

We introduce the *latent_status* to capture any edge that has a Remove message destined for it. More precisely, define the latent_status of edge e by: At step 0, for every edge e, latent_status(e) is False. latent_status(e) becomes True at step i if, at step i, an unchosen edge e' changes its chosen_status to True because of receipt of a Search message ("search", *sender*, *path*) where $sender = e'$ and e is the edge with maximum weight in *path*. latent_status(e) becomes False at step j if, at step j, edge e changes its chosen_status to False because of receipt of a Remove message ("remove", *path*) where e is the maximum weight edge of *path*. Define Latent_Set(i) to be the set $\{e \mid \text{In Config(i), latent\_status}(e) = True\}$.

Lemma 5.2.4 *If G is not a tree, then for all steps $i > S_8$, Chosen_Set$(i) \setminus$ Latent_Set(i) is a proper subset of E.*

Proof: Since Chosen_Set(i) is connected for any $i > S_3$, by Lemma 5.2.3, the only way for an unchosen edge to become chosen is line 20, which also adds an edge to Latent_Set. The only way for Latent_Set to lose an edge is if that edge changes to unchosen.

Suppose Chosen_Set$(3M) = E$ and Latent_Set$(3M) = \emptyset$. Since G is not a tree, there exists at least one cycle. If there is any Search message in the network, it will propagate to this cycle and generate a Remove message destined for the edge with maximum weight, say e, in that cycle by some step $i \leq S_4$. So $e \in$ Latent_Set(i). Otherwise, some edge of the cycle will time-out within time $3M$, generate a find_cycle message, and detect the cycle in at most M additional time. So by some step $j \leq S_8$, $e \in$ Latent_Set(j).

Therefore, once Chosen_Set$(i) \setminus$ Latent_Set(i) is a proper subset of E for $i \geq S_8$, it remains a proper subset for all $j > i$. ■

Lemma 5.2.5 *If G is not a tree, no chosen edge can time-out after S_8.*

Proof: By Lemma 5.2.4, there is always an unchosen edge or a latent edge in the network. Once a chosen edge has reset its timer, a latent edge will become unchosen and an unchosen edge will time-out and propagate a Search message, before a chosen edge can again time-out, because the chosen edge sets its timer to 3 times that required for any message to reach it. Because the chosen set is connected, the Search message will reach any chosen edge and will cause it to reset its timer before timing out. ■

Let $e_1, e_2, \ldots, e_m$ be the edges of G sorted in order of increasing weight. Let $\hat{E}$ be the subset of E consisting of edges of $MST(G)$. Define $k(s)$ be the smallest integer in $\{1, \ldots, m\}$ such that, $\forall i > k(s)$, $e_i \in$ Chosen_Set(s) if and only if $e_i \in \hat{E}$.

Observe that $k(s)$ is the index of the maximum weight edge such that the predicate $(e_{k(s)} \in$ Chosen_Set$(s))$ differs from $(e_{k(s)} \in \hat{E})$. The proof proceeds by showing that after step S_8, we have the safety property that the index $k(s)$ never increases, and the progress property that integer $k(s)$ eventually decreases. Then we will be able to conclude that eventually $k(s)$ must be 0, implying that the chosen set is the edge set of MST. All the remaining lemmas are implicitly intended to apply after step S_8.

Lemma 5.2.6 *For every edge e in $\hat{E}$, if $e \in Chosen\_Set(s)$ then $e \in Chosen\_Set(s'), \forall s' > s$.*

Proof: The only way that edge status changes from chosen to unchosen is by receipt of a Remove message, which can only remove the edge with maximum weight in some cycle. By Claim 3.1.3, no such edge can be in $\hat{E}$. ■

Lemma 5.2.7 $\hat{E} \subseteq Chosen\_Set(s)$.

Proof: By Lemma 5.2.6, $\forall e \in \hat{E}$ and $e \in$ Chosen_Set(s), e stays in Chosen_Set(s') for $s' > s$. If $\exists e \in \hat{E}$ and $e \notin$ Chosen_Set(s), $e = \langle u, v\rangle$ will time-out and initiate its Search message at one end-point, say u. By Lemma 5.2.3, $(V,$Chosen_Set$(s))$ is connected and spans the network. So some copy of e's Search message will return to its v end-point. By Claim 3.1.3, there must exist e' in the path travelled by the Search message, with larger weight than e. So e becomes a chosen edge. ■

Lemma 5.2.8 *$k(s)$ is non-increasing.*

Proof: By the definition of $k(s)$, if $i > k(s)$ and $e_i \in$ Chosen_Set(s), then $e_i \in \hat{E}$. By Lemma 5.2.6, for every $s' > s$ $e_i \in$ Chosen_Set(s').

By the definition of $k(s)$, If $i > k(s)$ and $e_i \notin$ Chosen_Set(s) then $e_i \notin \hat{E}$. Suppose $\exists s' > s$, $e_i \in$ Chosen_Set(s'). Only line 5 or line 19 can change chosen_status of e_i from unchosen to chosen. Line 5 is impossible after time S_3 by Lemma 5.2.3. So e_i received its own Search message ("search", e_i, *path*) indicating the maximum weight in *path* is a chosen edge e_j other than e_i. Since wt$(e_j) >$ wt(e_i), $j > i$ because indexes of edges are by increasing weight. Hence $j > k(s)$ and thus e_j is chosen implies $e_j \in \hat{E}$. But e_j is also a maximum edge in a cycle, contradicting Claim 3.1.3. ■

Lemma 5.2.9 *If $k(s) \geq 1$, then $\exists s' > s$ such that $k(s') < k(s)$.*

Proof: By the definition of $k(s)$, $e_{k(s)} \in$ Chosen_Set(s) if and only if $e_{k(s)} \notin \hat{E}$. By Lemma 5.2.7, $\hat{E} \subset$ Chosen_Set(s), so chosen_status$(e_{k(s)}) = True$ in Config(s) and $e_{k(s)} \notin \hat{E}$. Consider the unique fnd_cyl$(\hat{E}, e_{k(s)}) = (e_{k(s)}, e_{\alpha_1}, \ldots, e_{\alpha_l})$ of edges in $\hat{E}$ and the edge $e_{k(s)}$. Again by Lemma 5.2.7, $e_{\alpha_i} \in$ Chosen_Set(s) for each $1 \leq i \leq l$. So $e_{k(s)}, e_{\alpha_1}, \ldots, e_{\alpha_l}$ is a cycle of chosen edges in Config(s). In this cycle $e_{k(s)}$ must be heaviest, because by Claim 3.1.3 no edge in $\hat{E}$ can be the heaviest of any cycle of G. This cycle will be detected by some Search message that traverses it, and $e_{k(s)}$ will be removed from the chosen set at some subsequent step s'. Thus in Config(s'), chosen_status$(e_{k(s)}) =$ False. So by Lemma 5.2.8, $k(s') < k(s)$. ■

Lemma 5.2.10 *If $(V, Chosen\_Set(s))$ is a minimum spanning tree, then for all $s' > s$, $(V, Chosen\_Set(s'))$ is a minimum spanning tree.*

Proof: When Chosen_Set(s) is a minimum spanning tree, there is no cycle of chosen edges and there is a path of chosen edges between every pair of vertices. Every unchosen edge is the largest edge in the cycle consisting of itself and the path of chosen edges between its end-points. So in Self_Stabilizing_MST, the unchosen edges keep timing out and sending Search messages. Each Search message returns to its initiator with the information that the unchosen initiator is the edge with maximum weight in the path traversed. By Lemma 5.2.5 chosen edges do not time-out, so there are no Find_cycle messages generated. So the unchosen edge is passive until it next times-out. ■

Lemmas 5.2.8, 5.2.9 and 5.2.10 combine to show that our algorithm is correct.

Theorem 5.2.1 *Algorithm Self_Stabilizing_MST is a self-stabilizing solution for the minimum spanning tree problem on message-passing networks.*

5.3 Complexity Analysis

The complexity analysis of the self-stabilizing MST algorithm is not straightforward. However the upper bound of the time complexity of Self_Stabilizing_MSTcan be derived from the proof of correctness. Recall that the time complexity of self-stabilizing MST on a network G is defined to be the maximum time over all executions and over all initial configuration for the system to converge to a legitimate configuration. The legitimate configuration requires chosen_status(e) = True if and only if $e \in MST(G)$.

Recall that M is the maximum safetime for all the edges in the network.

Theorem 5.3.1 *The upper bound of time complexity for self-stabilizing MST algorithm is* $O\big((m-n)M\big)$ *for any graph with n nodes and m edges.*

Proof: In algorithm Self_Stabilizing_MST, within $3M$ at least one unchosen edge can be determined successfully whether it is in Chosen_Set. From the proof of correctness, after $8M$ time, the edge set $\hat{E}$ of MST is a subset of the Chosen_Set by Lemma 5.2.7. Furthermore, within the next each repetition, the largest $e \in$ Chosen_Set $\setminus \hat{E}$ becomes unchosen and stays unchosen. So the maximum time taken for the executions before stabilization is $\big(3(m-n)+8\big)M$. Thus an upper bound on the time complexity for the system to converge to a legitimate configuration is $O\big((m-n)M\big)$. ■

CHAPTER 6

Model Conversion

6.1 Simulation

We have presented distributed MST algorithms in the altered graph model where edges represent processores and nodes provide channels between neighbouring edge-processores. This chapter describes how the original network, G, where nodes are processores and edges represent message-passing channels, can simulate the algorithm designed for altered(G).

Since we assume that the processoror in the network G have distinct identifiers, one of the end-points of each edge can be selected to simulate the edge-processoror. The selection could be done by simply choosing the end-point with the larger identifier. The end-point with the larger identifier is charged with executing all the actions of the edge; the end-point with the smaller identifier is passive and only passes on messages of that edge.

Once nodes are selected to simulate edges, it is likely that some nodes will simulate more than one of their adjacent edges and that other node may simulates no edges. Say a node is *in Charge* of an edge if it is assigned to simulate that edge. Every node has an incoming simulator processor and an outgoing simulator processor. When a node v receives a message m from edge e, it passes m to the incoming simulator. If v is in charge of e, the incoming simulator simulates the processing of m by e and passes the resulting messages to its outgoing simulator. If v is not in charge of e, it simplely passes m to its outgoing simulator. When the outgoing simulator of v receives a message m' for delivery to edge

e' then if v is in charge of e', the outgoing simulator simulates the processing of m by e' and passes the resulting message to the other end-point of e'. If v is not in charge of e', the outgoing simulator simply passed m' to the other end-point of e'.

Our altered graph model assumed that an edge had access to the status of its neighbours edges. In the node simulation just this would require a node to have access to the status of other nodes up to a distance of at most two. For example, suppose node u is in charge of edge $\langle u, v\rangle$ and node w is in charge of edge $\langle v, w\rangle$. Node u's simulation of $\langle u, v\rangle$ would require that node u has access to the status$\langle v, w\rangle$ which is stored at of w, which is distance two away. However in the message passing network this information is not immediately available. Let AL_1 to be a self-stabilizing local update algorithm that is to gather information on the topology up to a radius of two. Our final simulator is achieve by the fair composite of AL_1 with the simulation AL_2 just describes and given in detail below.

Let $\langle p', p, w\rangle$ represent edge (p', p) with weight w. We use the notational convention that for all edges in all messages, the edge is listed as $\langle p, q\rangle$ if the message travels in the direction from p to q. For any edge (p, p'), if $p > p'$, then p will store all local variables wt(p', p), $timer(p', p)$, chosen_status(p', p) and $search\_sent(p', p)$. Also every processor p maintains an array of the estimated status of each processores that is at most two links away.

The self-stabilizing end-point simulating MST algorithm AL_2 is derived by translating Self_Stabilizing_MST based on the correct knowledge of the topology up to a radius of two.

Procedure for processor p:

`Upon` $timer(p', p)$ `time-out:` for any $p' < p$

1. If $\left(\neg\text{chosen\_status}(p', p)\right) \wedge \left(\neg search\_sent(p', p)\right)$ Then
2. For $p'' \in N(p) \setminus \{p'\}$ Do
3. If $(p > p'')$ Then
4. If $\left(\text{chosen\_status}(p, p'')\right)$ Then
5. `reset_timer` $timer(p, p'')$;
6. `send` $\left(\left(\text{“search”}, < p', p >, [< p, p'', w(p, p'') >]\right), p''\right)$;

Else
  `send` $((\text{“}\texttt{search}\text{”}, < p', p >, \emptyset), p'')$;
$search\_sent(p', p) \leftarrow$ True ;
Elseif $(\neg \text{chosen\_status}(p', p)) \wedge (search\_sent(p', p))$ Then
  chosen_status$(p', p) \leftarrow$ True;
Elseif $(\text{chosen\_status}(p', p))$ Then
  For $p'' \in N(p) \setminus \{p'\}$ Do
    If $(p > p'')$ Then
      If $(\text{chosen\_status}(p, p''))$ Then
        `reset_timer` $timer(p, p'')$;
        `send` $\Big((\text{“}\texttt{find\_cycle}\text{”}, [< p, p'', w(p, p'') >]), p''\Big)$;
    Else
      `send` $((\text{“}\texttt{find\_cycle}\text{”}, \emptyset), p'')$;
  `reset_timer` $timer(p', p)$;

`Upon receipt of` $(\text{“}\texttt{search}\text{”}, sender, path)$ `from` p'
If $(p > p') \wedge (< p', p > \neq sender)$ Then
  If $(\text{chosen\_status}(p', p))$ Then
    $path \leftarrow path \oplus < p', p, w(p', p) >$;
    `reset_timer` $timer(p', p)$;
    If $(\forall p'' \in N(P) \setminus \{p'\}, < p, p'', w(p, p'') > \notin path)$ Then
      For $p'' \in N(p) \setminus \{p'\}$ Do
        If $(p > p'')$ Then
          If $(\text{chosen\_status}(p, p'')) \wedge (< p, p'', w(p, p'') > \neq sender)$ Then
            `reset_timer` $timer(p, p'')$;
            If $(\forall \hat{p} \in N(p'') \setminus \{p\}, < p'', \hat{p}, w(p'', \hat{p}) > \notin path)$ Then
              `send` $\Big((\text{“}\texttt{search}\text{”}, sender, path \oplus < p, p'', w(p, p'') >), p''\Big)$;
            Else
              Let $path = list_1 \oplus list_2$ when $\texttt{head}(list_2) = < p'', \hat{p}, w(p'', \hat{p}) >$

$\texttt{send}\ \Big((\text{``}\texttt{remove}\text{''}, list_2),\ \ \texttt{second}\big(< p'', \hat{p}, w(p'', \hat{p}) >\big)\Big);$

Elseif $\big(\neg\text{chosen\_status}(p, p'')\big) \wedge \big(< p, p'', w(p, p'') >= \texttt{head}(path)\big)$ Then

$search\_sent(p, p'') \leftarrow \text{False};$

If $\big(\texttt{max\_weight}(path) > \text{wt}(p, p'')\big)$ Then

$\text{chosen\_status}(p, p'') \leftarrow \text{True};$

$\texttt{send}\ \Big((\text{``}\texttt{remove}\text{''}, path), \texttt{second}\big(\texttt{head}(path)\big)\Big);$

Else

$\texttt{send}\ \big((\text{``}\texttt{search}\text{''}, sender, path), p''\big);$

Else

Let $path = list_1 \oplus list_2$ when $\texttt{head}(list_2) =< p, p'', w(p, p'') >$

$\texttt{send}\ \Big((\text{``}\texttt{remove}\text{''}, list_2),\ \ \texttt{second}\big(< p, p'', w(p, p'') >\big)\Big);$

Elseif $\big(\neg\text{chosen\_status}(p', p)\big) \wedge \big(< p', p, w(p', p) >= sender\big)$ Then

$search\_sent(p', p) \leftarrow \text{False};$

If $\big(\texttt{max\_weight}(path) > \text{wt}(p', p)\big)$ Then

$\text{chosen\_status}(p', p) \leftarrow \text{True};$

$\texttt{send}\ \big((\text{``}\texttt{remove}\text{''}, path),\ \texttt{second}\ (\ \texttt{head}(path))\big);$

Elseif $\big(p < p'\big)$ Then

For $p'' \in N(p) \setminus \{p'\}$ Do

If $(p > p'')$ Then

If $\big(\text{chosen\_status}(p, p'')\big) \wedge \big(< p, p'', w(p, p'') > \neq \texttt{head}(path)\ \big)$ Then

$\texttt{reset\_timer}\ timer(p, p'');$

If $\big(\forall \hat{p} \in N(p'') \setminus \{p\}, < p'', \hat{p}, w(p'', \hat{p}) > \notin path\big)$ Then

$\texttt{send}\ \big((\text{``}\texttt{search}\text{''}, sender, path \oplus < p, p'', w(p, p'') >), p''\big);$

Else

Let $path = list_1 \oplus list_2$ when $\texttt{head}(list_2) =< p'', \hat{p}, w(p'', \hat{p}) >$

$\texttt{send}\ \big((\text{``}\texttt{remove}\text{''}, list_2),\ \ \texttt{second}(< p'', \hat{p}, w(p, \hat{p}) >)\big);$

Elseif $\big(\neg\text{chosen\_status}(p, p'')\big) \wedge \big(< p, p'', w(p, p'') >= sender\big)$ Then

$search\_sent(p, p'') \leftarrow \text{False};$

If $\left(\texttt{max\_weight}(path) > \mathrm{wt}(p, p'')\right)$ Then
chosen_status$(p, p'') \leftarrow$ True;
$\texttt{send}\big((\text{“}\texttt{remove}\text{”}, path), \texttt{second}(\texttt{head}(path)\big);$
Else
$\texttt{send}\ \big((\text{“}\texttt{search}\text{”}, sender, path), p''\big);$

$\texttt{Upon receipt of}$ (“remove”, $path$) from p'
If $(p > p')$ Then
If $\left(\mathrm{wt}(p', p) \neq max(path)\right)$ Then
If $p'' > p$ s.t. $< p, p'' w(p, p'') >= \texttt{head}\big(\texttt{tail}(path)\big);$
$\texttt{send}\big((\text{“}\texttt{remove}\text{”}, path), p''\big);$
Elseif $\left(\mathrm{wt}(p, p'') \neq max(path)\right)$ Then
$\texttt{send}\Big(\big(\text{“}\texttt{remove}\text{”}, \texttt{tail}(path)\big), p''\Big);$
Else
chosen_status$(p, p'') \leftarrow$ False;
$search\_sent(p, p'') \leftarrow$ False;
Else
chosen_status$(p', p) \leftarrow$ False;
$search\_sent(p', p) \leftarrow$ False;

$\texttt{Upon receipt of}$ (“find_cycle”, path) $\texttt{from}$ p'
If $\left(p > p'\right)$ Then
If $\left(\text{chosen\_status}(p', p)\right)$ Then
$path \leftarrow path \oplus < p', p, w(p', p) >;$
$\texttt{reset\_timer}\ timer(p', p);$
If $\left(\forall p'' \in N(P) \setminus \{p'\}, < p, p'', w(p, p'') > \notin path\right)$ Then
For $p'' \in N(p) \setminus \{p'\}$ Do
If $(p > p'')$ Then
If $\left(\text{chosen\_status}(p, p'')\right) \wedge \left(< p, p'', w(p, p'') > \neq \texttt{head}(path)\right)$ Then

reset_timer $timer(p,p'')$;
If $\left(\forall \hat{p} \in N(p'') \setminus \{p\}, < p'', \hat{p}, w(p'',\hat{p}) > \notin path\right)$ Then
send $\left((\text{“find\_cycle”}, path \oplus < p, p'', w(p,p'') >), p''\right)$;
Else
Let $path = list_1 \oplus list_2$ when $\texttt{head}(list_2) = < p'', \hat{p}, w(p'',\hat{p}) >$
send $\left((\text{“remove”}, list_2),\ \ \texttt{second}(< p'', \hat{p}, w(p'',\hat{p}) >)\right)$;
Else
send $\left((\text{“find\_cycle”}, path), p''\right)$;
Else
Let $path = list_1 \oplus list_2$ when $\texttt{head}(list_2) = < p, p'', w(p,p'') >$
send $\left((\text{“remove”}, list_2),\ \ \texttt{second}(< p, p'', w(p,p'') >)\right)$;
Else
For $p'' \in N(p) \setminus \{p'\}$ Do
If $(p > p'')$ Then
If $\left(\text{chosen\_status}(p'',p)\right) \wedge \left(< p, p'', w(p,p'') > \neq \texttt{head}(path)\right)$ Then
reset_timer $timer(p,p'')$;
If $\left(\forall \hat{p} \in N(p'') \setminus \{p\}, < p'', \hat{p}, w(p'',\hat{p}) > \notin path\right)$ Then
send $\Big((\text{“find\_cycle”}, path \oplus < p, p'', w(p,p'') >), p''\Big)$;
Else
Let $path = list_1 \oplus list_2$ when $\texttt{head}(list_2) = < p'', \hat{p}, w(p'',\hat{p}) >$
send $\Big((\text{“remove”}, list_2),\ \ \texttt{second}\big(< p'', \hat{p}, w(p'',\hat{p}) > \big)\Big)$;
Else
send $\left((\text{“find\_cycle”}, path), p''\right)$;

The self-stabilizing local update algorithm AL_1 is as follows:

Procedure for processor p:

Upon $timer(p)$ time-out
propagate $\left(\text{“update”}, \text{info}(p), 1\right)$

Upon receipt of $\left(\text{“}\texttt{update}\text{”}, \text{info}(p'), 1\right)$

$local(p') \leftarrow \text{info}(p')$

propagate $\left(\text{“}\texttt{update}\text{”}, \text{info}(p'), 2\right)$

Upon receipt of $\left(\text{“}\texttt{update}\text{”}, \text{info}(p'), 2\right)$

$local(p') \leftarrow \text{info}(p')$

The final algorithm is constructed from the fair composition of the self-stabilizing end-point simulation algorithm AL_2 of the edge-processor algorithm Self_Stabilizing_MST and the self-stabilizing local update algorithm, AL_1.

The proof of correctness of the self-stabilizing fair composition are provide in Section 2.7,of Shlomi Dolev's Self-Stabilization [6]. The theorem states:" Assume that AL_2 is self-stabilizing for a task T_2 given task T_1. If AL_1 is self-stabilizing for T_1, then the fair composition of AL_1 and AL_2 is self-stabilizing for T_2."

The final algorithm executes AL_1 and AL_2 alternatively, that is executing one step of AL_1 and then one step AL_2. The task T_1 makes sure that each node p updates its information about p' where p' is at distance one or two from p. So AL_1 for task T_1 only modifies the portion of p's states *local*(p'). The task T_2 is to construct the minimum spanning tree based on each node's states. So AL_2 of p for the task T_2 modifies its chosen_status(p).

CHAPTER 7

Conclusions

7.1 Summary of Contributions

This thesis presents two new algorithms for distributed MST on the message passing model. The first, Basic_MST, does not stabilize to the MST of the network from any initial configuration; rather, it converts any valid spanning tree configuration to the minimum spanning tree in a self-stabilizing fashion. This algorithm can be used to maintain a minimum spanning tree in a network that can grow by the addition of edges and nodes and can have edge weight changed dynamically. The algorithm Basic_MST also provides intuition for the more general case.

There are few self-stabilizing algorithms written directly for the message passing model. The algorithm Self_Stabilizing_MST is the first self-stabilizing algorithm that solves the minimum spanning tree problem on the message passing model. When the system is started from any initial configuration, the algorithm Self_Stabilizing_MST constructs a minimum spanning tree in a bounded number of steps. The algorithm applies to fully dynamic systems as long as safetime is not violated.

Unlike many self-stabilizing algorithm for distributed networks, neither Basic_MST or Self_Stabilizing_MST require a distinguished leader node in the network. This gives additional flexibility. For example algorithms that require a leader cannot be fully dynamic.

Significantly different proofs of convergence are needed for the two algorithms even though the second algorithm builds upon the ideas used in the first. Specially, the technique used in the proof of the correctness of the second algorithm is considered as one of contribution of this thesis. A new potential function is obtained that provides a measure of how far the current Chosen_Set is from the edge set of the minimum spanning tree. We show that the value of this function never increases and eventually decreases.

The presentation of algorithms Basic_MST and Self_Stabilizing_MST were simplified by describing them from the "edge-processor" perspective where edges rather than nodes were assumed to drive the computation. The final message-passing algorithm is obtain by translating from the edge driven to node driven setting, and then applying fair composition with a self-stabilizing local topology update algorithm.

Both algorithms differ markedly from the approach of the well-known distributed Minimum spanning tree algorithm (GHS) of Gallager, Humblet and Spira [9], because they are driven by the unchosen edges rather than attempting to construct a forest of trees that merge over time.

7.2 Observations, Discussion and Future Directions

Several observations lead to further questions concerning both algorithms and models for self-stabilization. These point to areas of potential future work.

The introduction of this thesis claimed that the Internet might provide one network where Self_Stabilizing_MST could be applied. The Internet, however, is so large that at any time there is likely to be some fault in it somewhere. Thus, it is highly unlikely that following a fault there would be a fault-free interval as least as big as the stabilization time, especially given the large worst-case time to stabilization of our algorithms. However, the repairs in Basic_MST and Self_Stabilizing_MST proceed in a distributed fashion and are typically quite independent. A fault in one part of a large network will only effect those parts of the spanning tree that might "see" that fault. For example, suppose a spanning tree edge fails by erroneously becoming unchosen due to a fault, or disappearing due to a dynamic network change. Then, only those spanning tree paths that use this edge are affected. In the Internet most connections follow approximately physically direct paths (routing from New York to Boston does not go via London), and due to caching and replication, a great deal of the traffic is relatively local. Thus most of the network spanning tree will continue to function without noticing the fault. Similar arguments can be made to defend the behaviour of the algorithms as adequate in the case of other faults or dynamic changes. Because there is no dependence on a root that must coordinate the revisions, the repairs to the spanning tree can proceed independently and "typically" locally. Note however, that there is no general local detection and correction claim that can be proved for either Basic_MST or Self_Stabilizing_MST.

Perhaps the lack of minimum spanning tree solutions is because a generalization of GHS to the self-stabilizing setting is not apparent. The GHS algorithm resembles a distributed version of Krushkal's algorithm. It maintains a spanning forest, the components of which are merged in a controlled way via minimum outgoing edges until there is only one component, which is the MST of the network. The algorithm relies heavily on the invariant that selected edges are cycle-free, which we do not see how to maintain in the self-stabilizing setting when specific edges (of small weight) must be selected. Algorithm Self_Stabilizing_MST differs markedly from the GHS kind of approach. The computation is driven by non-tree edges. It is more akin to local error detection than to forest growing.

Algorithm Self_Stabilizing_MST is correct for a completely dynamic network. Also if we assume that weights of edges are stored in unstable storage, the algorithm will still be correct. Weights do not need to be static or to reside in stable storage. If communication links are broken down or extra links are added into the network, the algorithm will still construct a minimum spanning tree. Algorithm Basic_MSTcan be used to maintain a minimum spanning tree in a network that has edges and nodes added or edges' weight changed dynamically. However, in contrast to Self_Stabilizing_MST, it can fail if an edge or a node is removed from the network dynamically. Suppose a chosen edge is removed from the network. This would result in a forest of two trees of chosen edges. We see no simply way to determine which unchosen edge should become chosen to maintain a spanning tree, as would be required by algorithm Basic_MST.

Basic_MST and Self_Stabilizing_MST work for network with distinct identifiers. They also work with distinct edge identifiers. Thus they still work if edges have distinct weights since these can serve as identifiers. However on an anonymous network without the guarantee of distinct weights, the algorithm can fail. Dolve [6] provides an algorithm that assigns distinct identifiers to an anonymous network. This could be composed with our Self_Stabilizing_MST algorithm to give a randomized self-stabilizing algorithm for MST on an anonymous message-passing network. It remains open whether a randomized algorithm could be used to give a more efficient solution than the algorithm presented in this thesis when the network has distinct identifies. It is also possible that on anonymous networks a more efficient solution could be found by further exploiting randomization.

To convert to the network model, we select one of the end-points of each edge to simulate that edge processor. If the selection of nodes to simulate edges is not well balanced, computation bottlenecks can arise. For example, if the end-point with largest identifier is selected for each edge and there are locally maximum identifiers of high degree in the network, then these nodes will be overworked. This in turn can slow down the communication and damage the whole purpose of the concurrency. We would like to find an algorithm that distributes the work load move evenly by more carefully tuning this assignment. The algorithm might switch the assignment to place the smaller ID in charge of processing or randomly assign one of the end-points to do the processing. Ideally this would be a dynamic and self-stabilizing assignment.

We assume the network has size n. Without this assumption, will the algorithms still be correct? The answer is yes provided some other knowledge is available. The network size is used only to set the time-out so knowledge of network size could be replaced by maximum time for a message to travel through all the processors.

Setting the safetime for each edge could present a challenging trade-off in some cases. Before stabilization, time-outs trigger some essential repair mechanisms in situations that would otherwise be deadlocked. After stabilization, no errors are detected so a processor does nothing until a time-out causes it to restart error detection. Thus inflated safetimes slow convergence in some situations, but reduce message traffic after stabilization. As can be seen from the algorithm, the unstable configurations that trigger time-outs are quite specialized and may be highly unlikely to occur during stabilization in some applications. In this case it may be advantageous to choose rather large time-out values. Further work including some simulation studies are necessary to determine appropriate safetimes for particular applications.

Our algorithms permit different safetime settings for each edge, which may be convenient since agreement does not need to be enforced. However, we do see how to exploit this possibility for efficiency while strictly maintaining self-stabilization. If faults caused by premature time-outs are tolerable, it may be reasonable to set safetime for each edge so that it is only likely to be safe rather than guaranteed. In this case it may be useful to exploit the possibility of different time-out intervals for different edges, by setting them to reflect the time required for a message to travel the edge's fundamental cycle instead of any simple path in the network.

In the introduction, we mention there are other models for the network communication. Can we use the ideas that help us design the Self_Stabilizing_MST to construct a self-stabilizing minimum spanning tree under read/write atomicity on a link-register model?

The technique to convert Basic_MST to a correct algorithm even for bounded channels was discussed in Chapter 4. However, in Self_Stabilizing_MST, there is a local variable search_sent. The purpose of this variable is to determine whether the chosen graph is disconnected. If we throw the Search messages away as we did in Basic_MST to deal with the bounded channels, then the variable search_sent could cause thrashing between procedure that changes edge status to chosen because the edge has evidence that the Chosen_Set is disconnected, and the procedure that changes status to unchosen because the edge collected evidence of a cycle in the chosen set. In this case when channels are bounded, the algorithm will fail to converge to the legitimate configuration.

The Search messages of Basic_MST are only $\Theta(\log n + \log w)$ bits each where n is the number of nodes in the network and w is largest weight in the network. However both Replace messages and Crowned messages record a path of up to n identifiers. So they have maximum size $\Theta\big(n(\log n + \log w)\big)$. If the search and replace procedures were proceeding sequentially these long messages would not be required. This is because propagation of a Replace message by an unchosen edge would reach its destination along a unique path of chosen edges, and the propagation of a crown response would travel the same path in reverse order of still chosen edges. However, because of concurrency, it is possible that the return path for the Crowned message is no longer composed entirely of chosen edges. So a propagation only along chosen edges risks losing the Crowned message and thus disconnecting the Target_Set. For algorithm Self_Stabilizing_MST, all three messages types can be long, because each contains a list of edge descriptions. Thus their size is $\Theta(n(\log n + \log w))$ bits. In this case the messages record paths of edges and their weights in order to detect cycles and to avoid the previously described problem. It remains an open problem whether a Self_Stabilizing_MST can be achieved using message with size at most $O(\log n + \log w)$ bits.

Both algorithms Basic_MST and Self_Stabilizing_MST create a lot of messages some of which can be very long. An important practical contribution would be to reduce the length and number of messages.

Bibliography

[1] G Antonoiu and PK Srimani. A self-stabilizing distributed algorithm to construct an arbitrary spanning tree of a connected graph. *Computers and Mathematics with Applications*, 30:1–7, 1995.

[2] G Antonoiu and PK Srimani. Distributed self-stabilizing algorithm for minimum spanning tree construction. In *Euro-par'97 Parallel Processing, Proceedings LNCS:1300*, pages 480–487. Springer-Verlag, 1997.

[3] Mikhail J Atallah. *Algorithms and Theory of Computation Hanndbook*. CRC Press LLC, 1999.

[4] NS Chen, HP Yu, and ST Huang. A self-stabilizing algorithm for constructing spanning trees. *Information Processing Letters*, 39:147–151, 1991.

[5] EW Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17:643–644, 1974.

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

[7] 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.

[8] S Dolev, A Israeli, and S Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7:3–16, 1993.

[9] R Gallager, P Humblet, and P Spira. A distributed algorithm for minimum weight spanning trees. *ACM Trans. on Prog. Lang. and Systems*, 5(1):66 – 77, 1983.

[10] MG Gouda and N Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40:448–458, 1991.

[11] ST Huang and NS Chen. A self-stabilizing algorithm for constructing breadth-first trees. *Information Processing Letters*, 41:109–117, 1992.

[12] JLW Kessels. An exerciae in proving self-stabilization with a variant funtion. *Inform. Process. Lett.*, 29:39 – 42, 1988.

[13] L Lamport. Solved problems, unsolved problems and non-problems in concurrency, invited address. In *PODC84 Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, pages 63–67, 1983.

[14] B M Maggs and S A Plotkin. Minimum spanning tree as a path finding problem. *Information Processing Letters*, 2:291–293, 1988.