12-2010

Self-stabilizing group membership protocol

Mahesh Subedi
University of Nevada, Las Vegas

SELF-STABALIZING GROUP MEMBERSHIP PROTOCOL

by

Mahesh Subedi

Bachelor of Engineering in Computer Engineering
Institute of Engineering, Tribhuvan University
2005

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science in Computer Science
School of Computer Science
Howard R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
December 2010

Mahesh Subedi

entitled

Self-Stabilizing Group Membership Protocol

be accepted in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science
School of Computer Science

Atjoy K. Datta, Committee Co-chair

John Minor, Committee Co-chair

Lawrence L. Larmore, Committee Member

Emma E. Regentova, Graduate Faculty Representative

Ronald Smith, Ph. D., Vice President for Research and Graduate Studies
and Dean of the Graduate College

December 2010

ABSTRACT

Self-Stabilizing Group Membership Protocol

by

Mahesh Subedi

Dr. Ajoy K. Datta, Examination Committee Chair
Professor of Computer Science
University of Nevada, Las Vegas

In this thesis, we consider the problem of partitioning a network into groups of bounded diameter.

Given a network of processes X and a constant D, the *group partition problem* is the problem of finding a D-partition of X, that is, a partition of X into disjoint connected subgraphs, which we call *groups*, each of diameter no greater than D. The *minimal group partition problem* is to find a D-partition $\{\mathrm{G}_1, \dots \mathrm{G}_\mathrm{m}\}$ of X such that no two groups can be combined; that is, for any G_i and G_j, where $i \neq j$, either $G_i \cup G_j$ is disconnected or $G_i \cup G_j$ has diameter greater than D.

In this thesis, a silent self-stabilizing asynchronous distributed algorithm is given for the minimal group partition problem in a network with unique IDs, using the composite model of computation. The algorithm is correct under the *unfair daemon*.

It is known that finding a D-partition of minimum cardinality of a network is $\mathcal{NP}$-complete. In the special case that X is the unit disk graph in the plane, the algorithm presented in this thesis is $O(D)$-competitive, that is, the number of groups in the partition constructed by the algorithm is $O(D)$ times the number of groups in the minimum D-partition.

Our method is to first construct a breadth-first search (BFS) tree for X, then find a maximal independent set (MIS) of X. Using the MIS and the BFS tree, an initial D-partition is constructed, after which groups are merged with adjacent groups until no more mergers are possible. The resulting D-partition is minimal.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to sincerely thank my advisor and graduate coordinator, Dr. Ajoy K. Datta for chairing my committee and advising this work. I am indebted to him for his guidance throughout this thesis work as well as for the invaluable support and guidance during my masters program. I would like to thank Dr. Lawrence L. Larmore for his invaluable help on the technical portion of the thesis and for participating on my committee.

I would like to thank Dr. John Minor and Dr. Emma E. Regentova for their participation in my committee.

I would like to thank the Department of Computer Science for their financial support through a Graduate Teaching Assistantship. I would like to thank all the professors with whom I worked as their teaching assistant during my entire UNLV period.

Finally I would like to thank my parents, who always miss me from other side of the earth, for their love and encouragement in difficult times.

CHAPTER 1

INTRODUCTION

The network topology of wireless ad hoc networks is highly dynamic and random. Nodes within such networks should be able to self-organize and maintain any logical communication infrastructure. Also, frequent changes in topology are hard to predict. Since mobile ad hoc networks are based on wireless links, they are more prone to message loss, and can experience higher delays and jitter, than fixed networks.

In addition to this, because of the highly dynamic nature of mobile ad hoc networks, any service running on top of these networks must be reliable. A group membership approach can help maintain reliability by providing a cluster of nodes over the network that complies with the properties required by the service using this network. Clusters of nodes within the network partition this network while adhering to the given problem constraints. Computing the maximum diameter of the network is one of the most important requirements of applications running on top of group membership protocols. Applications running on top of a group membership protocol leverage the management of execution context dynamics and node mobility by using this membership protocol. Group membership provides various functionalities like collaborative editing, providing fault tolerance, sharing computational load, etc.

A group management protocol in mobile ad hoc networks requires a number of design constraints and choices. Group constraints can be set according to the application that uses the underlying group membership service. These group constraints can be view size, diameter of the view, geographical positions of the view members, or some integrity and/or security constraints.

Beside the constraints required by the application running above the group management service, the protocol itself must be distributed and self-stabilizing to achieve fault tolerance. The group management protocol must be the same for each node running the protocol, independent of the underlying network or configurations. There should not be any centralized node to manage group membership. This helps achieve fault tolerance and load balancing in the network. Every distributed system is prone to various failures including node failures, memory corruption etc. The failure can be permanent, e.g. node failure, or temporary, e.g. memory corruption. The distributed system, regardless of the current state, should be guaranteed to recover to a legal configuration in a finite number of steps, and remain in the legal state until another fault occurs. Also, aside from overcoming faults, the protocol must overcome any churn, i.e. change in topology or any new appearance or disappearance of a node, in the network. Another important property of wireless ad hoc networks is the efficiency of the protocol. The overhead of group membership management must be low. The amount of message sending and receiving required, and the time required to achieve self-stabilization, must be minimum. This is critical in mobile wireless networks due to limited resources, specifically power constraints.

1.1 Contributions

We present a silent self-stabilizing distributed algorithm, in the composite model of computation, for the group membership or partition problem. Our algorithm works under the unfair daemon, and has a competitiveness of *O(d_max)* in the planar disk graph case. The time complexity of our algorithm is $O\left(\frac{n^2 diam}{d\_max^2}\right)$, where n is the number of *processes* in the network and *diam* is the diameter of the network. The space complexity of our algorithm is *O(H)* for each process, where *H* is the maximum cardinality of (*d_max+1)*-neighborhood of any process. Our algorithm is constructed using a new technique for combining distributed self-stabilizing algorithms.

1.2 Outline

In Chapter 2, we give an overview of the distributed systems, mobile ad hoc networks and group membership problem in general. We discuss the related background work on membership management protocols. In Chapter 3, we describe the model of computation used in the thesis and discuss distributed networks and dynamic arrays. Then we formally define the problem specification of the thesis.

Combining two different distributed self-stabilizing algorithms is given in Chapter 4. Chapter 5 provides the overview of the algorithm followed by more detailed description of the algorithm. We then present different mode of incompatibility. The preprocessing module is described in Chapter 6. Computation of dist, BFS and MIS trees, beta and the computation of initial partition is covered in the subsequent sections of chapter 6.

Chapter 7 and 8 describe the main modules of the algorithm Front and Back respectively. In Section 7.1 we describe the computation of a dynamic array for each process. Section 7.2 describes the computation of dynamic array *grp_dist*[ ] for error-checking purpose. The neighbor groups of current process dynamic array border_dist[ ] is computed in section 7.3. Dynamic array *strong_cert*[ ] is computed to decide whether to merge or not to merge two groups, we describe in section 7.4. Computation of *bid, agree* and *merge_dist* followed by computation of *near* and *far* are described in subsequent sections.

Two modules of back, *weak_cert* and *merge,* are described in sections 8.1 and 8.2 respectively.

In Chapter 9, we discuss the error detection of the algorithm followed by complexities and competitiveness in Chapter 10 and 11 respectively.

Chapter 12 concludes the thesis.

CHAPTER 2

BACKGROUND

2.1 Distributed Systems

A distributed system is a communication network, or a collection of independent computers that appears to its users as a single coherent system. It can even be a single multitasking computer [14]. Although the processors in distributed systems are autonomous in nature, they may need to communicate with each other to coordinate their actions and achieve a reasonable level of cooperation [24]. In a distributed system, a program composed of executable statements is run by each computer. Each execution of a statement changes the computer's local memory content, and hence the state of the computer. Consequently, a distributed system is modeled as a set of *n* state machines that communicate with each other.

In a distributed system, there are mainly two models of communication between machines: message passing and shared memory. In the message passing model, machines communicate with each other by sending and receiving messages, whereas in the shared memory model, communication is carried out by writing to and reading from the shared memory.

2.3 Self-stabilizing Systems

Self-Stabilization is related to autonomic computing, which entails several "self-*" attributes like: self-organized [3], self-configuration, selfhealing, and self-maintaining [25]. According to [25], research in a self-* system is “a direct response to the shift from needing bigger, faster, stronger computer systems to the need for less human-intensive management of the systems currently available. System complexity has reached the point where administration generally costs more than hardware and software infrastructure.” The goals of the self-* systems are reduction of human administration and maintenance, and an increase of reliability, availability and performance.

In 1973, Dijkstra introduced the term self-stabilization into the world of computer science [13]. The concept of self-stabilization is one of fault-tolerance. Unfortunately, only a few people had become aware of its importance until Lamport endorsed this as “Dijkstra’s most brilliant work” and a “milestone in work on fault-tolerance” in his invited talk at the ACM Symposium on Principles of Distributed Computing in 1983. Today it is one of the most active areas of research in the field of computer science.

A system is considered self-stabilizing if, starting from any arbitrary state (possibly a fault state), it is guaranteed to converge to a legitimate state which satisfies its problem specification in a finite number of steps. Once it converges to a legitimate state, it must stay in that legitimate state thereafter unless a fault occurs. With respect to behavior, it can also be defined as a system starting from an arbitrary state, reaching a state in finite time from which it starts behaving correctly according to its specification. Thus self-stabilization enables systems to recover from a transient fault automatically.

According to [6,5], self-stabilization can be defined in terms of two properties; closure and convergence. Closure means that if a system is in a correct (or legitimate) state, it is guaranteed to stay in a correct state, if no fault occurs. On the other hand, convergence means that starting from any arbitrary state, it is guaranteed that the system will eventually reach a correct state in finite steps. In order for a system to be self stabilizing, it must satisfy both of these properties.

Self –stabilization has been extensively studied in the area of network protocols. Protocols like routing, sensor networks, high-speed networks, and connection management are just a part of many applications of self-stabilization. Also, there exist many self-stabilizing distributed solutions for graph theory problems. Examples include spanning tree constructions, maximal matching, search structures, and graph coloring. Many self-stabilizing solutions for numerous classical distributed algorithms were also proposed. These include mutual exclusion, token circulation, leader election, distributed reset, termination detection, and propagation of information with feedback [14].

In the study of self-stabilization, several aspects of models have been considered, such as the following:

Inter process Communication: shared registers or message passing.

Fairness: weakly fair, strongly fair, or unfair.

Atomicity: composite or read/write atomicity.

Types of Daemon: central or distributed.

All in all, proving stabilization programs is quite challenging. Two techniques have been commonly used in research literature, convergence stair [19] and variant function [20] methods. Furthermore, many general methods of designing self-stabilizing programs have been proposed which include diffusing computation [4], silent stabilization [15], local stabilizer [1], local checking and local correction [8, 7], counter flushing [27], self-containment [18], snap-stabilization [11], super-stabilization [16], and transient fault detector [9].

Self-stabilization is a significant concept in the study of MANETs. Due to the dynamic nature of MANET topology, the protocols for setting up and organizing MANETs are desirable to be self-stabilizing.

2.3 Mobile Ad Hoc Networks

Mobile ad hoc networks are key to the evolution of wireless networks. Ad hoc networks are typically composed of equal nodes that communicate over wireless links without any central control. In this type of network, communication between two hosts is peer-to-peer, i.e., each host directly communicating with another connected host. Ad hoc networks have the same problems carried by wireless and mobile communications such as bandwidth optimization, power control, and transmission quality enhancements. Moreover, the multi-hop nature of ad hoc networks and lack of fixed infrastructures generates new research problems.

Mobile ad hoc networks in general are formed dynamically by an autonomous system of mobile nodes that are connected via wireless links without using the existing network infrastructure or centralized administration.

2.4 Related Work

Best effort group service[17] is a self-stabilizing dynamic distributed protocol which ensures that the diameter of each group is limited by an application specific maximum value (D-max). It tries to maintain existing groups unless strong topology changes occur. The continuity property allows an application running on top of best-effort group service to have a more consistent view while executing. To maintain continuity, the groups do not split unless required by diameter constraints.

In this protocol, any node whose neighbors within D-max hop distance are potential group members. By flooding messages in a neighborhood, a list of candidates can be discovered in D-max time. A current view members maintained by a node are then sent in the neighborhood. If the merging of the received list violates the diameter property, the list is ignored and the sender is marked as incompatible. Any addition of a new node in the group will be propagated to all the view members within D-max time. The arrival of this node is accepted only when this does not violate the diameter property. In the case of two members accepted by the two distant members of the view, one new member must leave the group to ensure that the existing group does not split. New members are added in view only after a D-max quarantine period to ensure they are not rejected by other members of the current view. When a node needs to leave the group to ensure the diameter constraint, the node with lowest priority is removed. If priority is not defined by the application using the membership service, is determined by node identity. Node identity is used to decide which node to remove.

CHAPTER 3

PRELIMINARIES

3.1 Model

We are given a connected undirected network, $G = (V, E)$ of $|V| = n$, where $n \geq 2$, and a distributed algorithm $\mathcal{A}$ on that network. Each process x has a unique ID, $x.id$. By an abuse of notation, we will identify each process with its ID.

A *self-stabilizing* [13, 14] system is guaranteed to converge to the intended behavior in finite time, regardless of the initial state of the system. In particular, a self-stabilizing distributed algorithm will eventually reach a *legitimate state* within finite time, regardless of its initial configuration, and will remain in a legitimate state forever. An algorithm is called *silent* if eventually all execution halts.

We use the composite atomicity model of computation, where each process has variables. Each process can read the values of its own and its neighbors', but can only write to its own variables. Each transition from a configuration to another, called a step of the algorithm, is driven by a *scheduler*, also called a *daemon*.

The *program* of each process consists of a finite set of actions of the following form: $< label > < informal\ name >< guard > \rightarrow < statement >$. For each action, the *label* is listed in the first column, and an informal name is listed in the second column. The third column (*guard*) contains a list of clauses, all of which must hold for the action to execute, and the fourth column contains the statement of the action. The *guard* of an action in the program of a process x is a Boolean expression involving the variables of x and its neighbors. The *statement* of an action of x updates one or more variables of process x. An action can be executed only if it is *enabled*, *i.e.,* its guard evaluates to true.

In the tables of programs, we assign a *priority*, a positive integer, to each action. The guard of each action is the conjunction of the clauses in the third column, together with the condition that no earlier (in terms of priority) action is enabled.

A process is said to be *enabled* if at least one of its actions is enabled. A step $\gamma_i \mapsto \gamma_{i+1}$ consists of one or more *enabled* process executing an *action*. The evaluations of all guards and executions of all statements of those actions are presumed to take place in one atomic step called *composite atomicity* [14]. All three of our algorithms are *uniform*, *i.e.,* every process has the same program.

When a process x executes the statement of an action, there could be neighbors of x that are executing statements during the same step. We specify that x uses the current values of its own variables (which could have just been changed during the current step), but old values of its neighbors' variables, *i.e.,* values before the current step.

We use the *distributed daemon*. If one or more processes are enabled, the daemon *selects* at least one of these enabled processes to execute an action. We also assume that the daemon is *unfair*, *i.e.,* that it need never select a given enabled process unless it becomes the only enabled process.

We define a *computation* to be a sequence of configurations $\gamma_p \mapsto \gamma_{p+1} \dots \mapsto \gamma_q$ such that each $\gamma_i \mapsto \gamma_{i+1}$ is a step.

We measure the time complexity in *rounds* [14]. The notion of *round* [14], captures the speed of the slowest process in an execution. We say that a finite computation $\rho = \gamma_p \mapsto \gamma_{p+1} \mapsto \cdots \mapsto \gamma_q$ is a *round* if the following two conditions hold:

1. Every process x that is enabled at γ_p either executes or becomes neutralized during some step of ρ. We say that a x is *neutralized* at a step $\gamma \mapsto \gamma'$ if x is enabled at γ and not enabled at γ', but x does not execute during that step.
2. The computation $\gamma_p \mapsto \cdots \mapsto \gamma_{q-1}$ does not satisfy condition 1.

We call a computation of positive length which fails to satisfy condition 1 an *incomplete round*.

We define the *round complexity* of a computation to be the number of disjoint rounds in the computation. More formally, we say that a computation $\gamma_p \mapsto \dots \mapsto \gamma_q$ has round complexity m if there exist indices $p = i_0 < i_1 < \cdots < i_{m-1} < q$ such that,

1. $\gamma_{i_{j-1}} \mapsto \cdots \mapsto \gamma_{i_j}$ is a round for all $1 \le j < m$,
2. $\gamma_{i_{m-1}} \mapsto \cdots \mapsto \gamma_q$ is either a round or an incomplete round.

We remark that an incomplete round could have infinite length, since the unfair daemon might never select an enabled process. But this cannot happen for the algorithms given in this paper. We will show that every computation of each of our algorithms is finite, *i.e.,* all the proposed algorithms in this thesis "work" under the unfair daemon.

3.2 Network

We are given a network of n processes with unique IDs. $N(x)$ is the set of neighbors of a process x. $U(x) = N(x) \cup \{x\}$.

The length of a path is defined to be the number of edges in the path. The distance $d(x,y)$ between processes x and y is defined to be the smallest length of any path between x and y.

Define $H_k(x) = \{y: d(x, y) \leq k\}$, to be the *k-neighborhood* of x. Thus, $U(x) = H_1(x)$.

A *subgraph* of $X = (V, E)$ is a set of processes V together with a set E of links between those processes. We say that a subgraph $G = (V_G, E_G)$ is *full* if every link of X both of whose ends are processes of G is a link of G. By abuse of notation, we will write $x \in G$ to mean $x \in V_G$ if x is a process, or $e \in G$ to mean that $e \in E_G$ if e is a link.

If $x, y \in G$ are processes, define $d_G(x,y)$ to be the length of the shortest path which lies entirely in G between x and y. If there is no such path we define $d_G(x,y) = \infty$. We say that G is *disconnected* if there exist processes $x, y \in G$ such that $d_G(x,y) = \infty$; otherwise, we say G is *connected*. Note that $d(x, y) \leq d_G(x,y)$.

The *size* of a subgraph *G*, written *size(G),* is the cardinality (number of processes) of *G*. A *component* of a subgraph *G* is the maximal non-empty connected subgraph of *G*. A non-empty connected subgraph has exactly one component.

The *diameter* of a non-empty connected subgraph *G*, written *diam(G),* is defined to be the maximum length of the minimum length path through *G*, between any two processes of *G*, *i.e.*, *diam(G)* = max $\{ d_G (x, y : x, y \in G \}$.

3.3 Dynamic Arrays

In our algorithm, each process will have both simple and array variables. In each case, the range of an array variable is a set of process IDs. The values and ranges of the arrays can change, and the range is normally smaller than the set of all process IDs. Thus, array variables are sparse dynamic arrays.

We illustrate this with an example. Each process x will have an array variable $x.dist[\ ]$, in which it will store the distances to all processes within $d\_max + 1$ of x. Thus, eventually, $Range(x.dist[\ ]) = H_{d\_max+1}(x)$. Initially, x does not know the IDs of those processes. If we write $x.dist[y]$, we mean the value of $d(x,y)$ that x has in its memory, which may not be the correct value. If x does not have a value for $d(x,y)$, *i.e..,* $y \notin Range(x.dist[\ ])$, we write $x.dist[y] = \perp$, where "⊥" is the symbol for "null," or "undefined."

If we need to set $x.dist[y]$ to a value k, we write $x.dist[y] \leftarrow k$. If $x.dist$[y] was previously defined, the old value is simply overwritten, but if, previously, $x.dist[y] = \perp$, then y is added to $Range(x.dist[\ ])$ and then the value k is assigned. Similarly, if we write $x.dist[y] \leftarrow \perp$, and previously $x.dist[y]$ was defined, then y is deleted from the $Range(x.dist[\ ])$. Because of arbitrary initialization, the initial range of $x.dist[\ ]$ could contain IDs of processes that are not within the allowed distance, or even fictitious IDs. Techniques for implementation of sparse dynamic arrays are well-known, and we do not concern ourselves with the details of that implementation.

We allow a process to reassign all values of a dynamic array in a single step. For example, in Action A1 in Table 6.1, we allow x to update the values of $x.dist$[y] for any number of y in a single step.

3.4 Problem Specification

We are given a positive integer *d_max*. We define *partition* of X to be a set of disjoint subgraphs, $\{G_1, \ldots G_m\}$, called *groups*, whose union contains all process of X, such that $diam(G_i) \leq d\_max$ for all i. We say that a partition is *minimal* if no two adjacent groups can be combined into a set whose diameter is at most *d_max*. A minimal partition may not be minimum, and it is known that finding a minimum partition, one which has the smallest possible number of groups, is $\mathcal{NP}$-hard.

Our problem is to find a minimal partition of the network, such that each process knows the ID and the distance, in its group, of every process in its group. In this thesis, we give a silent self-stabilizing algorithm which solves the problem.

CHAPTER 4

COMBINING SELF-STABILIZING ALGORITHMS

We now consider the problem of combining distributed algorithms. The problem of constructing such a combination, which is trivial for sequential algorithms, is somewhat harder for distributed algorithms.

For example, suppose $\mathcal{A}$ and $\mathcal{B}$ are algorithms, which are concatenated, *i.e.,* combined sequentially, to form an algorithm which we call $\mathcal{A} + \mathcal{B}$. We will call $\mathcal{A}$ and $\mathcal{B}$ *modules* of the combined algorithm. $\mathcal{A} + \mathcal{B}$ consists of first executing $\mathcal{A}$, then executing $\mathcal{B}$, which uses the output of $\mathcal{A}$ as its input.

This construction is trivial in the sequential model, but not at all easy in the distributed model. For example, suppose that $\mathcal{A}$ and $\mathcal{B}$ are both self-stabilizing and silent. That is, from an arbitrary configuration, $\mathcal{A}$ always converges to a configuration that satisfies some intermediate predicate, and then halts; while from a configuration which satisfies that intermediate predicate, $\mathcal{B}$ always converges to a configuration that satisfies some final predicate, and then halts.

More formally, we define an instance of the SSS-concatenation, *i.e.,* self stabilizing and silent distributed algorithm concatenation, problem to consist of the following.

1. A network X of processes, where each process x has a set of variables. Let $States(x)$ be the set of states of x, as normally

defined in the composite atomicity model, *i.e.,* each state of x is a vector consisting of a value for each variable of x.

Let $\mathbb{C} = \prod_{x \in X}$ States (x), the set of configurations of the network. For any $x \in X$, let $\mathbb{C}|U(x) = \prod_{y \in U(x)}$ States (y), the *local configuration* of x.

2. Two sets of actions, which we call the set of $\mathcal{A}$-actions and the set of $\mathcal{B}$ -actions. If $\gamma, \gamma', \gamma'' \in \mathbb{C}$, we write $\gamma \underset{A}{\rightarrow} \gamma'$, $\gamma \underset{B}{\rightarrow} \gamma''$, if there is an $\mathcal{A}$-action, respectively $\mathcal{B}$-action, which changes γ to γ', respectively γ''. Similarly, we write $\gamma \xrightarrow[A]{*} \gamma'''$ if there is an $\mathcal{A}$-computation, *i.e.,* a sequence of $\mathcal{A}$-actions, which changes γ to γ''', and we define $\gamma \xrightarrow[B]{*} \gamma''''$ similarly.
3. A set of configurations $\mathbb{A} \subseteq \mathbb{C}$, the set of intermediate legitimate states, such that every maximal $\mathcal{A}$-computation ends at a configuration in $\mathbb{A}$. At a configuration in $\mathbb{A}$, no process is enabled to execute an $\mathcal{A}$-action.
4. A set of configurations $\mathbb{B} \subseteq \mathbb{C}$, the set of final legitimate states, such that every maximal $\mathcal{B}$-computation which starts in $\mathbb{A}$ ends at a configuration in $\mathbb{B}$. At a configuration in $\mathbb{B}$, no process is enabled to execute a $\mathcal{B}$-action.

A *solution* to the above instance is an SSS distributed algorithm which converges to $\mathbb{B}$. We will only consider solutions which are obtained by adding additional variables. More formally, all our solutions will have the following properties.

1. Each process has all the same original variables, in addition to some other variables, which we call augmentation variables, or S-variables.

   Let $\text{States}_S(x)$ be the set of states of the augmentation variables of a process x, and let $\mathbb{S} = \prod_{x \in X} \text{States}_S(x)$, the set of augmentation configurations of X. In the combined algorithm, the set of configurations is $\mathbb{C} \times \mathbb{S}$. Each configuration of X is thus an ordered pair (γ, σ), where $\gamma \in \mathbb{C}$ is what we call the *base configuration*, and $\sigma \in \mathbb{S}$ is the augmentation configuration.

2. A set of actions for the combined algorithm, such that every maximal computation of the combined algorithm is finite and ends at a configuration in $\mathbb{B} \times \mathbb{S}$.

Unfortunately, we have no solution for the SSS-concatenation problem in general. We do, however, have solutions in some simple cases which occur in practice.

4.1 The Nested SSS-Concatenation Problem

We need some additional notation. We write $\mathcal{A}$-$Enabled$(x), respectively $\mathcal{B}$-$Enabled(x)$, if a process x is enabled to execute an $\mathcal{A}$-action, respectively $\mathcal{B}$-action.

We define an instance of the *nested SSS-concatenation problem* to be an instance of the SSS-concatenation problem which satisfies the following additional conditions.

1. $\mathbb{B} \subseteq \mathbb{A}$
2. There is a subset of variables of each process, which we call $\mathcal{A}$-variables, such that
   (a) the predicate $\mathcal{A}$-Enabled(x) depends only on the values of the $\mathcal{A}$-variables of x and its neighbors,
   (b) no $\mathcal{B}$-action changes an $\mathcal{A}$-variable,
   (c) if $\gamma_0 \underset{B}{\rightarrow} \gamma_1 \underset{B}{\rightarrow} \ldots$ is a $\mathcal{B}$-computation, and if no process which executes during that computation is $\mathcal{A}$-enabled at the time it executes, then the computation is finite.
   Note: there is no guarantee that a maximal $\mathcal{B}$-computation that satisfies the above restriction terminates in $\mathbb{B}$, unless it begins in $\mathbb{A}$.

We can now implement $\mathcal{A} + \mathcal{B}$ by using priorities; a process x cannot execute a $\mathcal{B}$-action if it is enabled to execute an $\mathcal{A}$-action. We call this combination of algorithms *nested concatenation*.

Table 4.1: Actions of $\mathcal{A} + \mathcal{B}$ for Process x: Nested Legitimacy Sets

A1 Priority 1	$\mathcal{A}$	$\mathcal{A}$-Enabled(x)	$\longrightarrow$	x executes an $\mathcal{A}$-action
A1 Priority 2	$\mathcal{B}$	$\mathcal{B}$-Enabled(x)	$\longrightarrow$	x executes a $\mathcal{B}$-action

We illustrate the relation between the sets of configurations $\mathbb{A}$, $\mathbb{B}$, and $\mathbb{C}$, in Figure 4.1.

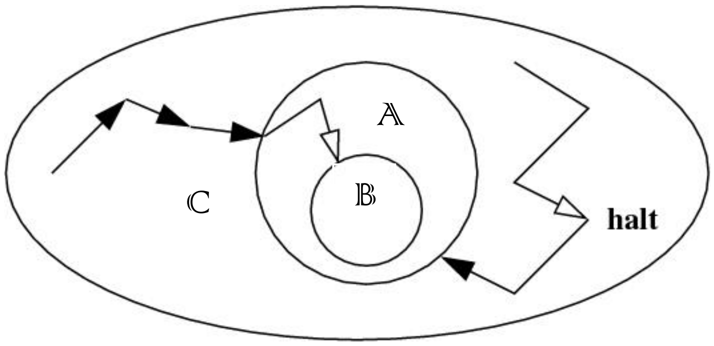


Figure 4.1 Relation between set of configurations
In concatenation, where legitimacy sets are nested, $\mathcal{A}$-Enabled is defined only in terms of $\mathcal{A}$-variables. $\mathcal{A}$-actions are shown as solid-headed arrows, while $\mathcal{B}$-actions are indicated with open heads. Any execution outside $\mathbb{A}$ consisting of only $\mathcal{B}$-actions is finite, provided $\mathcal{A}$-actions have priority over $\mathcal{B}$-actions.

Nested concatenation is used in the literature. For example, in [28], nested concatenation is used to construct the algorithm BFS-MIS which is used in this paper as a module for our algorithm. Also, in this thesis, we use nested concatenation to build the three main modules of our algorithm from submodules.

4.2 The Non-Nested Restricted SSS-Concatenation Problem

We now consider a somewhat less restricted special case of the SSS-concatenation problem.

We define an instance of the *non-nested restricted SSS-concatenation problem* to be an instance of the SSS-concatenation problem which satisfies the following additional conditions.

1. There is a set of configurations $\mathbb{D} \subseteq \mathbb{C}$ such that

   (a) $\mathbb{A} \subseteq \mathbb{D}$

   (b) $\mathbb{B} \subseteq \mathbb{D}$

   (c) Any $\mathcal{B}$-computation starting from any configuration in $\mathcal{D}$ is finite, and ends in $\mathbb{B}$.

2. There is a predicate $\mathcal{B}\_Error\ (x)$ defined for each process x such that any maximal $\mathcal{B}$ computation either ends in $\mathbb{B}$ or contains a configuration where $\mathcal{B}\_Error\ (x)$ holds for some process x.

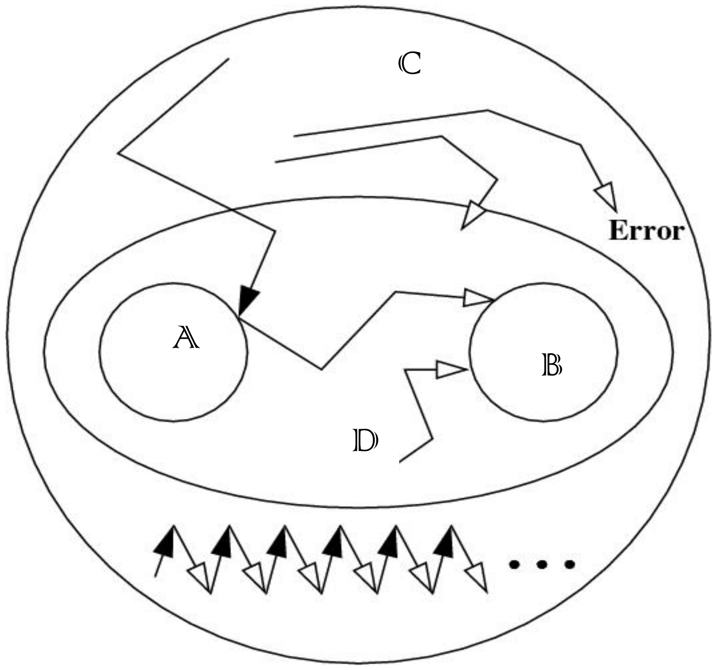


Figure 4.2 Non-Nested Restricted Concatenation Problem.
Actions of $\mathcal{A}$ are shown as solid-headed arrows, while actions of $\mathcal{B}$ are indicated with open heads. From anywhere, a computation of $\mathcal{A}$ leads to $\mathbb{A} \subseteq \mathbb{D}$. From anywhere inside $\mathbb{D}$, a computation of $\mathcal{B}$ leads to $\mathbb{B}$. Executions of actions of $\mathcal{B}$ outside of $\mathcal{D}$ are undesirable, and could slow down convergence of $\mathcal{A}$. Any computation of $\mathcal{B}$ eventually enters $\mathbb{D}$, or is detected as erroneous by some process, but a computation mixing actions of $\mathcal{A}$ and $\mathcal{B}$ could continue forever without entering or being detected as erroneous. (Although shown as disjoint in the figure, $\mathbb{A}$ and $\mathbb{B}$ could intersect.)

In order to construct the general concatenation $\mathcal{A} + \mathcal{B}$, we need to introduce additional variables and actions, and thus to expand the definition of a configuration.

1. We assume the existence of a self-stabilizing silent *leader election* algorithm(module) LE. We do not concern ourselves with the actions and variables of LE, other than the following requirements that must be met when LE is silent:

(a) There is a leader process.

(b) Each process x has a non-negative integer variable $x.level$, which is equal to the distance (*i.e.,* length of the shortest path) between x and the leader of its component.

For example, the algorithm given in [28] could be used for LE.

2. For any process x, define

   $Preds(x) = \{y \in N(x) : y.level = x.level - 1\}$ the *predecessors* of x.

   $Succs(x) = \{y \in N(x) : y.level = x.level + 1\}$ the *successors* of x.

3. The LE-configuration is defined to be the configuration of the network defined by considering only variables of LE. Let $\mathbb{LE}$ be the set of all LE-configurations, and let $\mathbb{L}$ be the set of all legitimate, *i.e.,* silent, configurations of LE.

4. Each process x has variables $x.color \in \{0, 1, 2, 3\}$ and $x.\mathrm{mode} \in \{\mathrm{A}, \mathrm{B}\}$, called the *color* and the *mode* of x.

   We define the *color-mode configuration* to be the configuration of X defined by considering only color and mode variables. Let $\mathbb{M}$ be the set of all color-mode configurations.

   Thus, $\mathbb{S} = \mathbb{LE} \times \mathbb{M}$, the set of augmentation configurations.

5. We define the *complete configuration* to be the ordered triple (γ, λ, μ), where γ is the base configuration, λ is the LE-configuration, and μ is the color-mode configuration of the network. Thus, the set of complete configurations of the network is $\mathbb{C} \times \mathbb{S} = \mathbb{C} \times \mathbb{LE} \times \mathbb{M}$.

6. We let LE-$Enabled(x)$ be the predicate defined using only the local LE-configuration of a process, which indicates that x is enabled to execute an action of LE.

We now give an overview of $\mathcal{A} + \mathcal{B}$ in the non-nested restricted case. LE-actions execute with highest priority, ignoring the local base and color-mode configurations. After LE is silent, the configuration lies in $\mathbb{C} \times \mathbb{L} \times \mathbb{M}$. The level values essentially define a BFS tree rooted at the leader. We will use that tree as a communication backbone to enforce the correct order of computations of $\mathcal{A}$-actions and $\mathcal{B}$-actions.

The problem we face in concatenating $\mathcal{A}$ and $\mathcal{B}$ is that, once $\mathcal{A}$ has become silent, the $\mathcal{B}$-actions could cause processes to once again become $\mathcal{A}$-enabled. This could result in an error, since the output variables of $\mathcal{A}$ could be merely temporary, intended to be altered when $\mathcal{B}$ executes. Our solution is to use $x.mode$ to indicate which of the two modules x is permitted to execute, and to use *color waves* to signal to processes that the execution of $\mathcal{A}$ is finished and they can change their mode from A to B.

We now explain in detail how the order of computation is enforced. If a process x detects any error (such as could be caused by the fact that an arbitrary initial configuration is permitted) $x.mode \leftarrow$A and $x.\mathrm{color} \leftarrow 0$. Each process remains in the color-mode state $(\mathrm{A}, 0)$ as long as it has not finished executing both LE and $\mathcal{A}$. When the root, *i.e.,* the leader elected by LE, detects that it is finished with both, it initiates a top-down color wave, changing all colors to 1, unless that wave is interrupted by the fact that not all calculations of LE and $\mathcal{A}$ are finished. This interruption can occur any number of times, but eventually, the color 1 wave will reach the leaves, and a convergecast wave begins changing the colors of all processes to 2.

It is possible that the color 2 wave will also be interrupted, since that wave could start at some leaves while calculations of $\mathcal{A}$ are continuing in other portions of the network. But, eventually, the leader will have color 2, and unless there is an error caused by the arbitrary initialization, all processes will have color 2 when the leader has color 2.

Finally, a top-down color 3 wave will start from the leader. Each process, while changing its color to 3, knows that (unless the configuration is in error) all calculations of $\mathcal{A}$ are finished throughout the network. When process and all its neighbors have color 3, it changes its mode to B, and is then is ready to execute actions of $\mathcal{B}$. These actions could cause a process to once again become $\mathcal{A}$-enabled, but that enablement will be ignored. Eventually, $\mathcal{B}$ will be silent, and thus $\mathcal{A} + \mathcal{B}$ will be silent.

We now list additional functions we need to implement $\mathcal{A} + \mathcal{B}$.

1. $Color\_Mode\_Error\,(x)$, a Boolean which means that one of the following holds:

   (a) $x.mode\ = \mathrm{B}$ and $y.clr \neq 3$ for some $y \in U(x)$.

   (b) $x.color\ = 3$ and $y.clr \in \{0,1\}$ for some $y \in U(x)$.

   (c) $x.color\ = 3$ and $y.color\ = 2$ for some $y \in Preds$ (x).

   Color-Mode error can only occur because of erroneous arbitrary initialization.

2. $Color\_Inversion(x)$, a Boolean which holds if one of the following holds:

   (a) $x.color\ = 2$ and $y.color\ = 0$ for some $y \in N(x)$.

   (b) $x.color\ = 1$ and $y.color\ = 0$ for some $y \in Preds\ (x)$.

   Color inversion is not an error; it merely indicates that some processes achieved local silence of $\mathcal{A}$ and LE while processes elsewhere were still executing $\mathcal{A}$-actions or LE-actions.

Table 4.2: Actions of $\mathcal{A} + \mathcal{B}$ in the Restricted Non-Nested Case for Process x

A1 Priority 1	LE	LE-*Enabled(x)*	$\longrightarrow$	x executes an LE-action $x.mode \leftarrow$A $x.color \leftarrow 0$
A2 Priority 2	$\mathcal{B}$-Error	$x.mode$ = B $\mathcal{B}$-Error (x)	$\longrightarrow$	$x.mode \leftarrow$ A $x.color \leftarrow 0$
A3 Priority 2	Color-Mode Error	$x.mode$ = B $Color\_Mode\_Error\ (x)$	$\longrightarrow$	$x.mode \leftarrow$ A $x.color \leftarrow 0$
A4 Priority 3	$\mathcal{A}$ Action	$\forall y \in U(x):\ y.mode$ = A $\mathcal{A}$-$Enabled(x)$	$\longrightarrow$	x executes an $\mathcal{A}$-action $x.color \leftarrow 0$
A5 Priority 3	$\mathcal{B}$ Action	$\forall y \in U(x):\ y.mode$ =B $\mathcal{B}$-$Enabled\ (x)$	$\longrightarrow$	x executes a $\mathcal{B}$-action
A6 Priority 4	Color Inversion	$x.\ mode$ = A $Color\_Inversion(x)$	$\longrightarrow$	$x.color \leftarrow 0$

A7	Broadcast	$x.color \ = c \in \{0, 2\}$	$\longrightarrow$	$x.color \ \leftarrow c + 1$
Priority 4	Color Wave	$x.mode \ = \mathrm{A}$		
		$\forall y \in Preds\ (x): \ y.color$		
		$= c + 1$		
		$\forall y \in Succs\ (x): y.color \ = c$		
		$\forall y \in N(x): \ y.color$		
		$\in \{c, c + 1\}$		
A8	Convergeca	$x.color \ = 1$	$\longrightarrow$	$x.color \ \leftarrow 2$
Priority 4	st Color	$x.mode \ = \mathrm{A}$		
	Wave	$\forall y \in Preds\ (x): y.color = 1$		
		$\forall y \in Succs\ (x): y.color \ = 2$		
		$\forall y \in N(x): \ y.color \ \in \{1, 2\}$		
A9	End $\mathcal{A}$	$x.mode \ = \mathrm{A}$	$\longrightarrow$	$x.mode \ = \mathrm{B}$
Priority 4	Start $\mathcal{B}$	$x.color \ = 3$		
		$\forall y \in N(x): y.color \ = 3$		

4.3 Combining Distributed Algorithms in a Loop

We now consider a much harder combination construction, which we need for our algorithm in this paper. We call this the *SSS-loop combination problem*. Once again, the sequential version of the problem is trivial. Suppose we are given modules $\mathcal{P}$, $\mathcal{A}$, and $\mathcal{B}$, and we wish to execute $\mathcal{P}$ first, followed by a loop which alternates execution of $\mathcal{A}$ and $\mathcal{B}$

until neither module is capable of further execution. We could encode this algorithm as follows:

Table 4.3: Sequential version of $\mathcal{P}$ + Loop($\mathcal{A}$, $\mathcal{B}$)

```
1: Execute $\mathcal{P}$ until it is finished
2: **repeat**
3:    Execute $\mathcal{A}$ until it is finished
4:    Execute $\mathcal{B}$ until it is finished
5: **until** neither $\mathcal{A}$ nor $\mathcal{B}$ can execute any more.
```

The SSS-loop combination problem is to design a self-stabilizing silent distributed algorithm which accomplishes the same task as the sequential algorithm given above. We define an instance of the problem to consist of the following.

1. Just as for the SSS-concatenation problem, we have a network X, where each process has variables, and $\mathbb{C}$ is set of configurations of the network.
2. Three sets of actions, which we call the set of $\mathcal{P}$-actions, the set of $\mathcal{A}$-actions, and the set of $\mathcal{B}$-actions.
3. Sets of configurations $\mathbb{P}$, $\mathbb{D}$, $\mathbb{E}$, $\mathbb{A}$, $\mathbb{B} \subseteq \mathbb{C}$, such that

   (a) $\mathbb{P} \subseteq \mathbb{D}$.

(b) $\mathbb{A}, \mathbb{B} \subseteq \mathbb{D} \cap \mathbb{E}$.

(c) $\mathbb{A} \cap \mathbb{B} \neq \emptyset$.

as illustrated in Figure 4.3, and such that

(a) No process is $\mathcal{P}$-enabled in $\mathbb{P}$.

(b) No process is $\mathcal{A}$-enabled in $\mathbb{A}$.

(c) No process is $\mathcal{B}$-enabled in $\mathbb{B}$.

(d) Every maximal $\mathcal{P}$-computation is finite and ends in $\mathbb{P}$.

(e) Every maximal $\mathcal{A}$-computation that begins in $\mathbb{D}$ stays in $\mathbb{D}$ and ends in $\mathbb{A}$.

(f) Every maximal B-computation that begins in E stays in E and ends in B.

4. Predicates $\mathcal{A}$_0k (x) $\mathcal{B}$_0k (x), computable by x, such that

   (a) Every maximal $\mathcal{A}$-computation either ends in $\mathbb{A}$ or contains a configuration in which $\neg\mathcal{A}$_0k (x) for at least one process x.

   (b) Every maximal $\mathcal{B}$-computation either ends in $\mathbb{B}$ or contains a configuration in which $\neg\mathcal{B}$_0k (x) for at least one process x.

5. Any alternating sequence of configurations of the form

$$\gamma_0 \xrightarrow[A]{*} \gamma_1 \xrightarrow[B]{*} \gamma_2 \xrightarrow[A]{*} \gamma_3 \xrightarrow[B]{*} \gamma_4 \cdots$$

   such that $\gamma_i \in \mathbb{A}$ if i is odd and $\gamma_i \in \mathbb{B}$ if i is even, is finite.

The purpose of this condition is to ensure that the combined algorithm eventually terminates.

Our task is to design a self-stabilizing silent distributed algorithm, $\mathcal{P}+\mathrm{LOOP}(\mathcal{A},\mathcal{B})$, which works under the unfair daemon, and which emulates the following computation:

1. Starting from any configuration in $\mathbb{C}$, execute $\mathcal{P}$-actions until the configuration reaches $\mathbb{P}$.
2. Execute the following loop until the configuration reaches $\mathbb{A} \cap \mathbb{B}$.

   (a) Execute $\mathcal{A}$ actions until the configuration reaches $\mathbb{A}$.

   (b) Execute $\mathcal{B}$ actions until the configuration reaches $\mathbb{B}$.

Figure 4.3 illustrates the desired computation. Our problem is to prevent processes from executing actions when they are not supposed to.

In order to solve the problem, we use augmentation variables in the same manner as in Section 4.2. Again, we use the variables of a leader election algorithm LE, as well as color variables $x.olor \in \{0,1,2,3\}$, and mode variables $x.mode \in \{\mathrm{P},\mathrm{A},\mathrm{B}\}$ for each process x.

We now give an overview of $\mathcal{P}+\mathrm{LOOP}(\mathcal{A},\mathcal{B})$. LE-actions execute with highest priority, ignoring the local base and color-mode configurations. After LE is silent, the configuration lies in $\mathbb{C}\times\mathbb{L}\times\mathbb{M}$. The level values essentially define a BFS tree rooted at the leader. We will use that tree as a communication backbone to enforce the correct order of computations of $\mathcal{P}$-actions, $\mathcal{A}$-actions, and $\mathcal{B}$-actions.

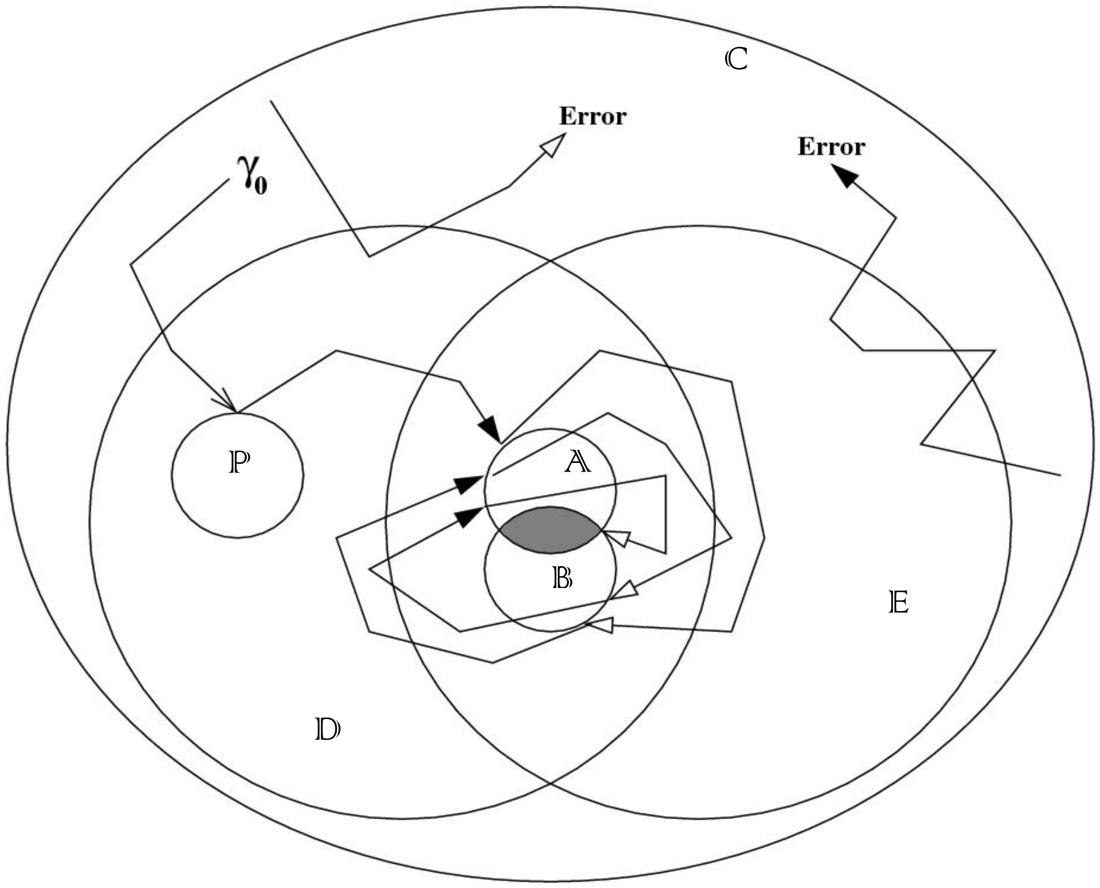


Figure 4.3 Loop Case
A computation of $\mathcal{A}$ starting outside $\mathbb{D}$, or a computation of $\mathcal{B}$ starting outside $\mathbb{E}$, could end in an error, which causes the mode to change to P. A complete execution of $\mathcal{P}$ + LOOP($\mathcal{A}$, $\mathcal{B}$), is also shown starting from γ_0. Initially, only $\mathcal{P}$ executes. When the configuration reaches $\mathbb{P}$, $\mathcal{A}$ executes until the configuration reaches $\mathbb{A}$. The algorithm then alternates between computations of $\mathcal{B}$ which reach $\mathbb{B}$ and computations of $\mathcal{A}$ reaching $\mathbb{A}$. When the configuration reaches $\mathbb{A} \cap \mathbb{B}$, the algorithm is silent.

The major problem we face is keeping each module from executing while another module is executing. We solve this problem using modes and color waves, using the same methods we used in Section 4.2.

In that section, we used color waves only during $\mathcal{A}$-executions. Once $\mathcal{B}$-execution began, the value of $\mathrm{x.color}$ remained 3 for all x. In $\mathcal{P} + \mathrm{LOOP}(\mathcal{A},\mathcal{B})$, on the other hand, colors are used for all three sets of actions. As before, the color of each process is 0 when it is executing, and then changes to 1, 2, and 3, in successive waves. When $x.color = 3$, then x knows that execution of the current module has finished, and can proceed to execute the next module.

We make use of the following predicates.

1. $\mathcal{P}\text{-}Enabled(x)$, meaning that x is enabled to execute an action of $\mathcal{P}$.
2. $\mathcal{A}\text{-}Enabled(x)$, meaning that x is enabled to execute an action of $\mathcal{A}$.
3. $\mathcal{B}\text{-}Enabled(x)$, meaning that x is enabled to execute an action of $\mathcal{B}$.
4. $Color\_Mode\_Error\,(x)$, a Boolean for $y \notin Preds\,(x)$, holds if the combination of colors and modes of x and its neighbors indicate the need to start the computation over. If $y \in Succs\,(x)$, the value of $Color\_Mode\_Error\,(x,y)$ is given in Table 4.4 otherwise, the value is given in Table 4.5. $Color\_Mode\_Error\,(x,y)$ is undefined if $y \in Preds\,(x)$.
5. $Color\_Inversion(x,y)$, a Boolean for $y \notin Preds\,(x)$, holds if no error has occurred, but x and y detect that one of them must revert its

color to 0. If $y \in Succs\ (x)$, the value of $Color\_Inversion(x, y)$ is given in Table 4.4; otherwise, the value is given in Table 4.5. $Color\_Inversion(x, y)$ is undefined if $y \in Preds\ (x)$.

6. $Can\_Switch\ (x)$ means that x is permitted to change mode in a normal manner, *i.e.,* not due to error. This predicate holds provided the following conditions hold.

   (a) $x.color\ = 3$

   (b) For all $y \in N(x)$, either $y.color\ = 3$ and $y.mode\ = x.mode$, or

   $$y.clr = 0 \text{ and } y.\ mode\ = \begin{cases} \mathrm{A} & if\ x.mode\ \in \{\mathrm{P}, \mathrm{B}\} \\ \mathrm{B} & if\ x.mode\ = \mathrm{A} \end{cases}$$

If $Can\_Switch\ (x)$ holds, then the color-mode configuration of x can change from (3, P) or (3, B) to (0, A), or from (3, A) to (0, B), as illustrated in Figure 4.4.

Table 4.4: Color Modes for $y \in Succs(x)$.
E denotes that $Color\_Mode\_Error(x)$ holds, I denotes that $ColorInversion(x, y)$ holds.

y.mode	*P*	*P*	*P*	*P*	*A*	*A*	*A*	*A*	*B*	*B*	*B*	*B*
y.color	0	1	2	3	0	1	2	3	0	1	2	3
x.mode = P *x.color=0*			*I*	*E*	*E*	*E*	*E*	*E*	*E*	*E*	*E*	*E*
x.mode = P *x.color=1*	*I*			*E*	*E*	*E*	*E*	*E*	*E*	*E*	*E*	*E*
x.mode = P *x.color=2*	*I*	*E*		*E*	*E*	*E*	*E*	*E*	*E*	*E*	*E*	*E*
x.mode = P *x.color=3*	*E*	*E*				*E*	*E*	*E*	*E*	*E*	*E*	*E*
x.mode = A *x.color=0*	*E*	*E*	*E*				*I*	*E*	*E*	*E*	*E*	
x.mode = A *x.color=1*	*E*	*E*	*E*	*E*	*I*			*E*	*E*	*E*	*E*	*E*
x.mode = A *x.color=2*	*E*	*E*	*E*	*E*	*I*	*E*		*E*	*E*	*E*	*E*	*E*
x.mode = A *x.color=3*	*E*	*E*	*E*	*E*	*E*	*E*				*E*	*E*	*E*
x.mode = B *x.color=0*	*E*	*E*	*E*	*E*	*E*	*E*	*E*				*I*	*E*
x.mode = B *x.color=1*	*E*	*E*	*E*	*E*	*E*	*E*	*E*	*E*	*I*			*E*
x.mode = B *x.color=2*	*E*	*E*	*E*	*E*	*E*	*E*	*E*	*E*	*I*	*E*		*E*
x.mode = B *x.color=3*	*E*	*E*	*E*	*E*		*E*	*E*	*E*	*E*	*E*		

Table 4.5: Color modes when $y \in N(x)$ and $y \notin Preds(x) \cup Sucs(x)$. E denotes that Color_Mode_Error(x) holds, I denotes that ColorInversion(x, y) holds.

y.mode	*P*	*P*	*P*	*P*	*A*	*A*	*A*	*A*	*B*	*B*	*B*	*B*
y.color	0	1	2	3	0	1	2	3	0	1	2	3
x.mode = P *x.color=0*			*I*	*E*	*E*	*E*	*E*	*E*	*E*	*E*	*E*	*E*
x.mode = P *x.color=1*				*E*	*E*	*E*	*E*	*E*	*E*	*E*	*E*	*E*
x.mode = P *x.color=2*	*I*				*E*	*E*	*E*	*E*	*E*	*E*	*E*	*E*
x.mode = P *x.color=3*	*E*	*E*				*E*	*E*	*E*	*E*	*E*	*E*	*E*
x.mode = A *x.color=0*	*E*	*E*	*E*				*I*	*E*	*E*	*E*	*E*	
x.mode = A *x.color=1*	*E*	*E*	*E*	*E*				*E*	*E*	*E*	*E*	*E*
x.mode = A *x.color=2*	*E*	*E*	*E*	*E*	*I*				*E*	*E*	*E*	*E*
x.mode = A *x.color=3*	*E*	*E*	*E*	*E*	*E*	*E*				*E*	*E*	*E*
x.mode = B *x.color=0*	*E*	*E*	*E*	*E*	*E*	*E*	*E*				*I*	*E*
x.mode = B *x.color=1*	*E*	*E*	*E*	*E*	*E*	*E*	*E*	*E*				*E*
x.mode = B *x.color=2*	*E*	*E*	*E*	*E*	*E*	*E*	*E*	*E*	*I*			
x.mode = B *x.color=3*	*E*	*E*	*E*	*E*	*E*	*E*	*E*	*E*	*E*	*E*		

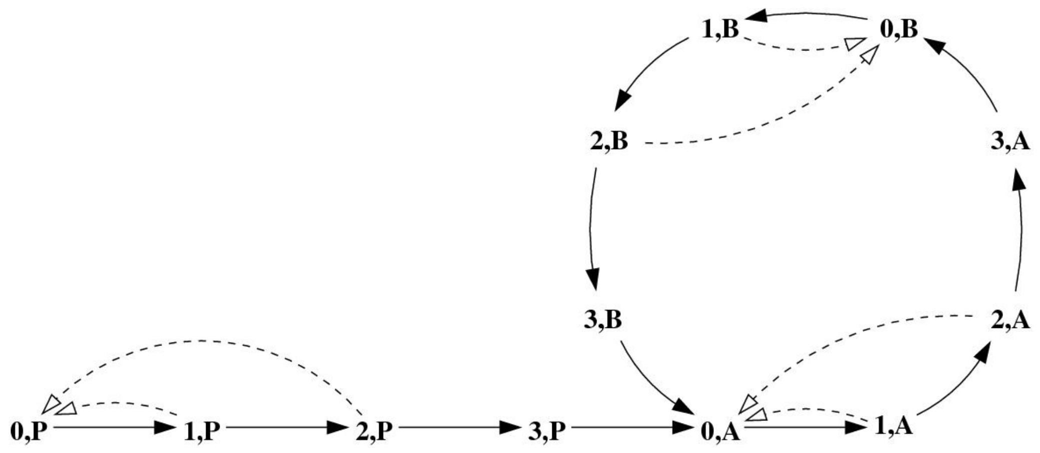


Figure 4.3 Normal progression of color-mode configurations in the absence of error.
Solid arrows represent broadcast or convergecast color waves, or normal switching of mode. Dashed arrows represent changes caused by either color inversion or by a process executing an action. In case of error, from anywhere in the figure, the color-mode configuration reverts to(0, P). Those changes are not indicated in the figure.

We give the actions of our implementation of $\mathcal{P}$ + Loop($\mathcal{A}$, $\mathcal{B}$) in Table

4.6

Table 4.6: Actions of $\mathcal{A} + \mathcal{B}$ in the Restricted Non-Nested Case for Process x

A1 Priority 1	LE	LE-*Enabled(x)*	$\longrightarrow$	x executes an LE-action
A2 Priority 2	Not in $\mathbb{D}$	$x.mode = \text{A}$ $\neg\mathcal{A}\_Ok\ (x)$	$\longrightarrow$	$x.mode \leftarrow \text{P}$ $x.color \leftarrow 0$
A3 Priority 2	Not in $\mathbb{E}$	$x.mode = \text{B}$ $\neg\mathcal{B}\_Ok\ (x)$	$\longrightarrow$	$x.mode \leftarrow \text{P}$ $x.color \leftarrow 0$
A4 Priority 2	Color Mode Error	$\exists y \in N(x) : Color\_Mode\_Error(x, y)$ or $Color\_\ Mode\ \_\ Error\ (y, x)$	$\longrightarrow$	$x.mode \leftarrow \text{P}$ $x.color \leftarrow 0$
A5 Priority 3	$\mathcal{A}$ Action	$\forall y \in U(x) :\ y.mode = \text{A}$ $\mathcal{A}$-$Enabled\ (x)$	$\longrightarrow$	x executes an $\mathcal{A}$-action
A6 Priority 3	$\mathcal{B}$ Action	$\forall y \in U(x) :\ y.mode = \text{B}$ $\mathcal{B}$-$Enabled\ (x)$	$\longrightarrow$	x executes a $\mathcal{B}$-action
A7 Priority 3	Color Inversion	$\forall y \in U(x) : Color\_Inversion(x, y)$ or $Color\_Inversion(\text{y}, x)$	$\longrightarrow$	$x.color \leftarrow 0$

A8	Broadcast	$x.color\ = c \in \{0,2\}$	$\longrightarrow\ x.color\ \leftarrow c+1$
Priority 4	Color Wave	$\forall y \in Preds\ (x) :\ y.color = c+1$	
		$\forall y \in Succs\ (x) : y.color\ = c$	
		$\forall y \in N(x) :\ y.color\ \in \{c, c+1\}$	
A9	Converge-cast	$x.color\ = 1$	$\longrightarrow\ x.color\ \leftarrow 2$
Priority 4	Color Wave	$\forall y \in Preds\ (x) : y.color = 1$	
		$\forall y \in Succs\ (x) : y.color\ = 2$	
		$\forall y \in N(x) :\ y.color\ \in \{1,2\}$	
A10	End $\mathcal{P}$	$x.mode\ = \mathrm{P}$	$\longrightarrow$ $\mathrm{x.mode}\ = \mathrm{A}$
Priority 4	Start $\mathcal{A}$	*Can_Switch(x)*	$\mathrm{x}.color = 0$
A11	End $\mathcal{B}$	$x.mode\ = \mathrm{B}$	$\longrightarrow$ $\mathrm{x.mode}\ = \mathrm{A}$
Priority 4	Start $\mathcal{A}$	*Can_Switch(x)*	$\mathrm{x}.color = 0$
A12	End $\mathcal{A}$	$x.mode\ = \mathrm{A}$	$\longrightarrow$ $\mathrm{x.mode}\ = \mathrm{B}$
Priority 4	Start $\mathcal{B}$	*Can_Switch(x)*	$\mathrm{x}.color = 0$

CHAPTER 5

PURPOSED ALGORITHM

5.1 Overview of the Algorithm

In this section, first we give an intuitive description of the algorithm. Our algorithm consists of two phases: preprocessing and merging. During the preprocessing phase, we create an initial partition. Each group of the initial partition (with the possible exception of just one group) contains at least $d\_max/2$ processes.

During the merging phase we merge groups in pairs. If $\{ G_1, \ldots G_p\}$ is a partition, we say that G_i and G_j are compatible if $G_i \cup G_j$ is connected and has diameter at most d_max. Otherwise, we say that G_i and G_j are incompatible. We identify three types of incompatibility. G_i and G_j could be not adjacent, G_i and G_j could be adjacent and strongly incompatible, or G_i and G_j could be adjacent and weakly incompatible.

The merging phase consists of a loop. During the first part of each iteration, each pair of adjacent groups decides whether to attempt to merge, or they will determine that they are incompatible. In the first case, progress toward a minimal partition has been made because there are fewer groups, and in the second case, progress has been made because that particular pair will not try to merge again. Eventually, every group will know that it is incompatible with every neighboring group, and thus the partition will be minimal.

5.2 Detailed Overview of the Algorithm

In this subsection, we give a top level description of the algorithm. Figure 5.1 illustrates the algorithm, where the boxes represent parts which will be separately described in subsequent subsections. The construction of the algorithm is done by concatenation, as explained in Section 4. In fact, our algorithm is precisely Preprocess + LOOP(Front,Back), as defined in Section 4.3, where Preprocess, Front, and Back are indicated by the outer boxes in Figure 5.1.

Two of those three processes are simple concatenations of subprocesses, following the paradigm explained in Section 4. We write

$\text{Preprocess} = \text{Comp}(dist) + \text{BFS} + \text{MIS} + \text{Comp}(\beta) + \text{Comp}(init\_leader) + \text{Comp}(leader)$

$\text{Front} = \text{Comp}(ldr) + \text{Comp}(grp_{dist}) + \text{Comp}(leader_{dist}) + Comp(strong\_cert) + \text{Comp}(bid) + \text{Comp}(agree) + \text{Comp}(merge\_dist)$

where $\text{Comp}(dist)$ is the module that computes $\text{x}.dist[\ ]$ for each x, etc.. The module Back is composed of two submodules, Merge and $\text{Comp}(weak\_cert)$. However, Back is not the concatenation of those two submodules. We will define the structure of Back explicitly in Section 8.

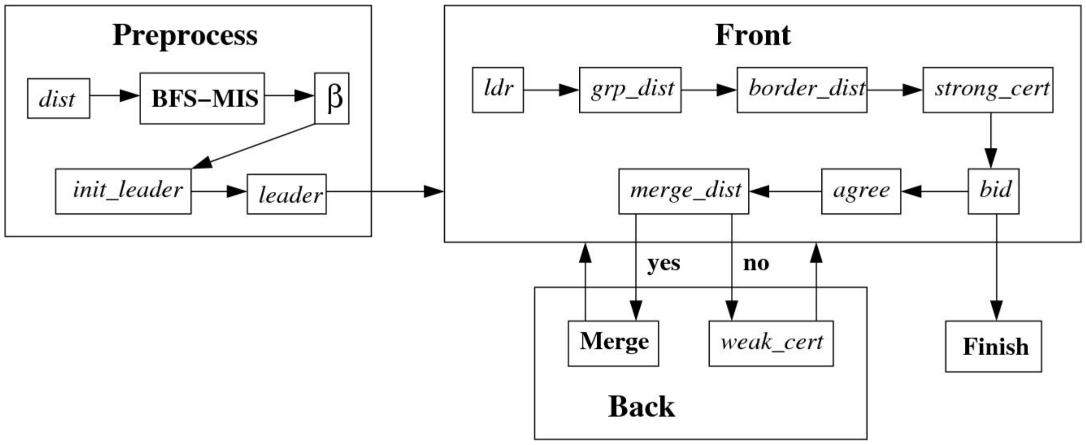


Figure 5.1 Normal flow of the algorithm.
The boxes indicate individual modules.

We now give a more detailed description of each of the submodules of our algorithm.

The module LE, which elects a leader for the network and computes $x.level$, the distance from x to the leader, for each process x, is not shown separately in Figure 5.3, since its job is taken over by the submodule Comp(BFS).

The module Preprocess, which plays the role of $\mathcal{P}$ as given in Section 4.3, consists of five submodules, as follows.

1. Comp($dist$), which computes the array variable $x.dist[\ ]$ for each process x. The correct value of $x.dist[y]$ is $d(x,y)$, provided that distance is at most $d\_max + 1$; otherwise, $x.dist[y] = \perp$. DIST is defined in Section 6.4. The values of $x.dist[\ ]$ are permanent, *i.e.,* when this submodule converges, they will never again be changed.

]

2. BFS-MIS, which elects a leader of X, and computes $\mathit{\Gamma}_{\mathrm{BFS}}$ and $\mathit{\Gamma}_{\mathrm{MIS}}$, the BFS tree and the MIS tree of X, respectively. Both trees are rooted at the leader, which we call *Root_BFS*. That module also constructs a maximal independent set, MIS, which consists of all processes at even levels in $\mathit{\Gamma}_{\mathrm{MIS}}$. BFS-MIS is taken from [28] and is described in section 6.2. The values of the variables computed by BFS-MIS are permanent.
3. Computation of *x.β,* an integer x.β $\in \{0, \dots, d\_\max\}$ for each x, in bottom up fashion on $\mathit{\Gamma}_{MIS}$, which guides the construction of the initial partition. The computation of *x.β* is described in Section 6.3
4. The next module computes the initial partition, *i.e*, the choice of $x.init\_leader$ for each x. The initial partition is in fact the minimum partition of the tree $\mathit{\Gamma}_{MIS}$, and every initial group, with the possible exception of the group containing *Root*_BFS, contains at least $d\_max/2$ processes, of which at least $(d\_max + 2)/4$ are in the maximal independent set.
5. Comp($leader$) simply executes $x.leader \leftarrow x.init\_leader$ for each process x. These values could change if x later executes the submodule Merge, which is part of the module Back; however, the values of $x.init\_leader$ are permanent.

The loop consists of two modules, Front and Back. Each of those modules has a number of variables that can change each time that module executes, but not during the execution of the other module.

Front is the simple concatenation of seven modules, using the technique given in Section 5:

1. Computation of the dynamic array $x.ldr[\ ]$ for all x. The correct value of $x.ldr[y]$ is $y.leader$ for all $y \in H_{dmax+1}$.
2. Computation of the dynamic array $x.grp\_dist[\ ]$ for all x. The correct value of $x.grp\_dist[y]$ is $d_{G(x)}(x,y)$ for all $y \in G(x)$, the current group which contains x.
3. Computation of the dynamic array $x.border\_dist[\ ]$ for all x. After convergence of that module, $border\_dist[\ell]$ is only defined if ℓ is the leader of a group which borders $G(x)$. The correct value of $x.grp\_dist[\ell]$ is $d_{G(x)}(x,y)$, where y is the nearest process of $G(x)$ which neighbors some member of $G(\ell)$.
4. Computation of the dynamic array $x.strong\_cert[\ ]$ for all x. After convergence of that module, $strong\_cert[\ell]$ is only defined if ℓ is the leader of a group which borders $G(x)$, and if $G(\ell)$ contains some process which has distance greater than d_max from some process in $G(x)$. The correct value of $x.strong\_cert[\ell]$ is the shortest distance, from x to some $y \in G(x)$ whose distance to some process in $G(\ell)$ is exactly $d\_max + 1$. If $x.strong\_cert[\ell] \neq \perp$ after convergence, the groups $G(x)$ and $G(\ell)$ will never be part of the same group, since the diameter of their union exceeds d_max.
5. Computation of the variable $x.bid$ for all x. The correct value of $x.bid$ is the the leader ℓ of a neighbor group which could possibly

merge with $G(x)$, meaning that $x.strong\_cert[\ell] =\perp$ and $x.weak\_cert[\ell] =\perp$, as we shall explain in Sections 8.12. If there are multiple such groups, $x.bid$ *is the minimum choice.* If there is no such group, $x.bid =\perp$ after the module converges.

If $x.bid = \ell$, then x has made a "bid" to merge $G(x)$ with $G(\ell)$. If, after convergence of Main, $x.bid =\perp$ for all x, then no more merging is possible, and the algorithm is silent.

6. Computation of x.$agree$ for all x. If $x.bid =\perp$, then the correct value of x.$agree$ is FALSE. Otherwise, the correct value of $x.agree$ is TRUE if, after convergence of Front, x.$bid = \ell$ and x.$agree =$ TRUE. In that case $\ell.bid = x.leader$, *i.e.,* each of the two groups has a bid to merge with the other. We call this situation a "mutual agreement to attempt to merge." During the next execution of Back, the two groups will merge if their union has diameter at most d_max.
7. Computation of $x.merge\_dist$. If $x.bid = \ell$ and $x.agree =$ TRUE, meaning that $G(x)$ has an agreement to attempt to merge with the neighboring group $G(\ell)$, then $x.merge\_dist[y]$ is computed for all $y \in G(x) \cup G(\ell)$. The value of $x.merge\_dist[y]$ is an integer in the range $0 \ldots 2\,d\_max + 1$, and its correct value is the length of the shortest path in $G(x) \cup G(\ell)$ from x to y.

Back consists of two submodules, but is not the concatenation of the submodules. Instead, the two submodules of Back are independent.

1. If G(x) has an agreement to merge with G(ℓ), and $u.merge\_dist[v] \le d\_max$ for all $u \in G(x)$ and all $v \in G(\ell)$, then $G(x)$ and $G(\ell)$ will merge.
2. On the other hand, if G(x) has an agreement to merge with G(ℓ), and there exist $u \in G(x)$ and $v \in G(\ell)$ such that $u.merge\_dist[v] = d\_max + 1$, then the two groups will not merge; instead, a weak certificate will be created to prevent G(x) and G(ℓ) from attempting to merge again.

5.3 Strong and Weak Incompatibility

We say that groups G_i and G_j are *strongly incompatible* if there exists processes x ∈ G_i and y ∈ G_j, where d(x, y) > *d_max*. In this case, G_i and G_j cannot be merged. But a stronger condition also holds: If $G_i \subseteq G'_k$ and $G_j \subseteq G'_l$ for some subsequent partition $G'_1, \dots G'_q$, then G'_k cannot be merged with G'_l. (See Figure 5.2)

If G_i and G_j are adjacent and not strongly incompatible, we say that they are *weakly incompatible* if $diam(G_i \cup G_j) > d\_max$. For example, in Figure 5.4, G(19) and G(23) are weakly incompatible.

Figures 5.3, 5.4, and 5.5 show various situations that can arise. In each of those figures, three groups are indicated with different shadings, and the leader of each group is indicated by a larger circle around the process. Note that there is no requirement that the leader be the process of smallest ID in the group. We let $d\_max = 7$ for all three examples.

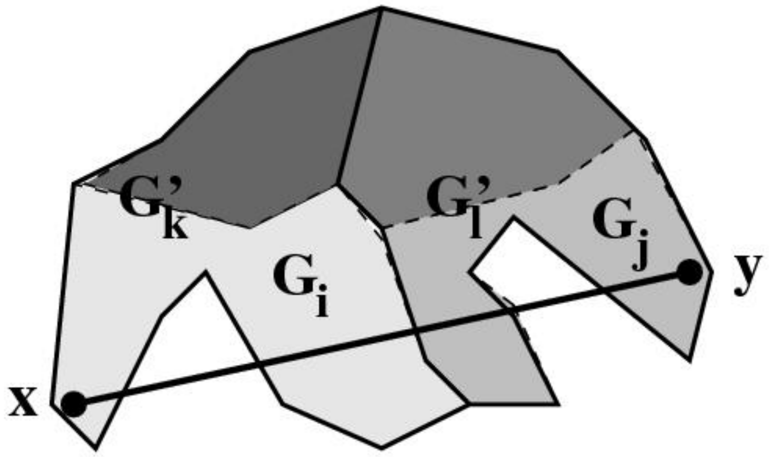


Figure 5.2 Strong incompatibility.
G_i and G_j are strongly incompatible, $G_i \subseteq G'_k$, and $G_j \subseteq G'_\ell$. Thus G'_j and G'_ℓ are strongly incompatible.

In Figure 5.3, the groups G(19) and G(23) are strongly incompatible to each other, because there are processes in those two groups which are more than 7 apart. For example, $d(91{,}50) = 8$. G(19) and G(23) will offer to merge with G(56). Using the "smallest leader ID" rule, G(56) will offer to merge with G(19). The groups G(19) and G(56) will then succeed in merging into a single group, which will be strongly incompatible with G(23). At that time a minimal partition is achieved.

In Figure 5.4, we show three groups, with leaders 19, 23, and 56. The groups G(19) and G(23) are not strongly incompatible, since $d(x, y) \leq 7$ for any $x \in G(19)$ and $y \in G(23)$. G(19) will offer to merge with G(23). If G(23) also offers to merge with G(19), then those two groups have a mutual agreement to try to merge. However that attempt will fail, since the diameter of the union G(19) ∪ G(23) is greater than 7. Both G(19) and G(23) will then remember that they are weakly incompatible.

Weak incompatibility may not survive merger with a third group. If, during the next iteration, G(19) and G(56) offer to merge with each other, they will succeed, creating a new group, which will now have leader 19, since we pick the smaller of the two leaders to be the new leader. At this point, G(23) is compatible with the new (larger) G(19), and if they offer to merge with each other, they will merge.

Figure 5.5 shows a situation where any two of three groups are weakly incompatible, but the union of all three groups would yield a group of diameter 7. Unfortunately, our algorithm is deadlocked in this situation, *i.e.*, none of the three will be merged with either of the others.

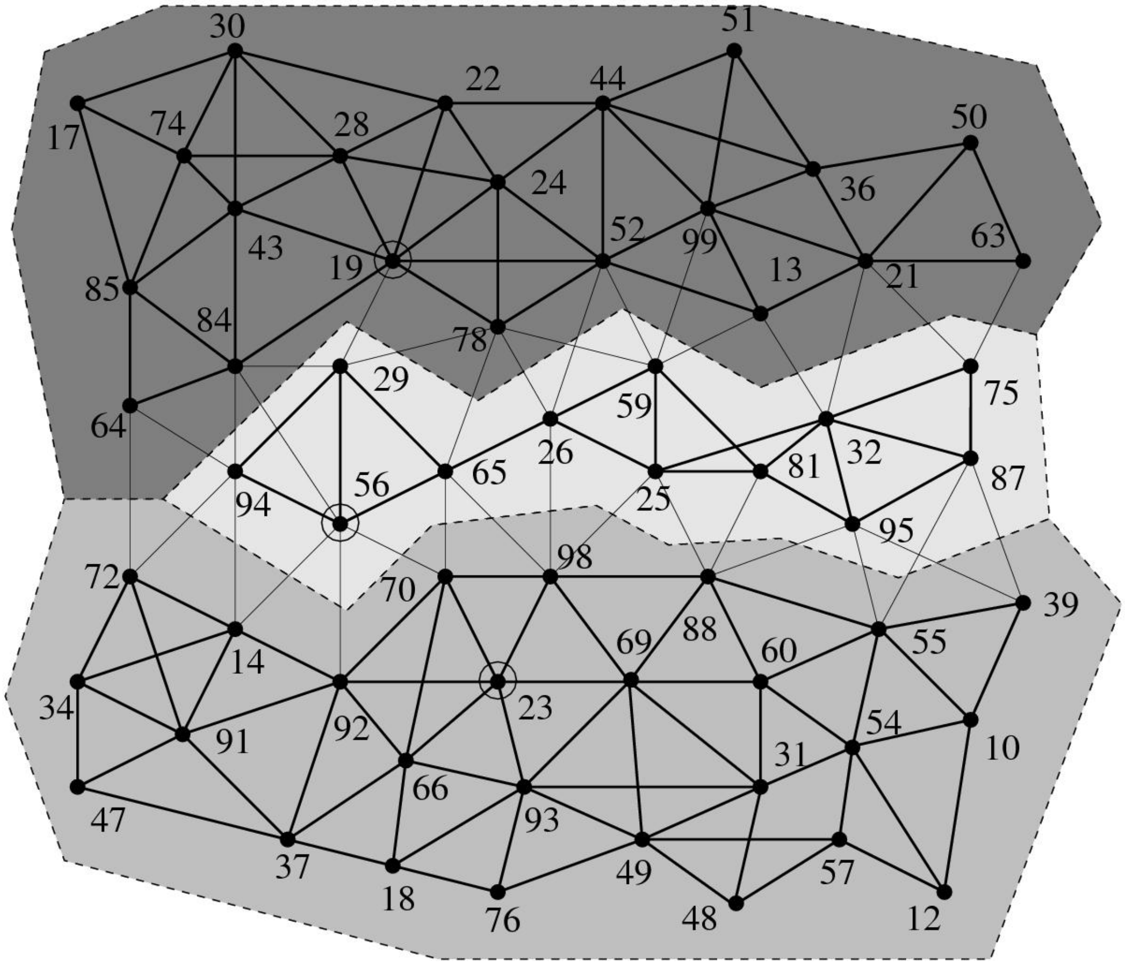


Figure 5.3 Strongly incompatible processes
Let *d_max* =7. *G(19)* and *G(23)* are strongly incompatible, but both are compatible with *G(56)*. If *G(19)* later merges with *G(56)*, the resulting group will still be strongly incompatible with *G(23)*.

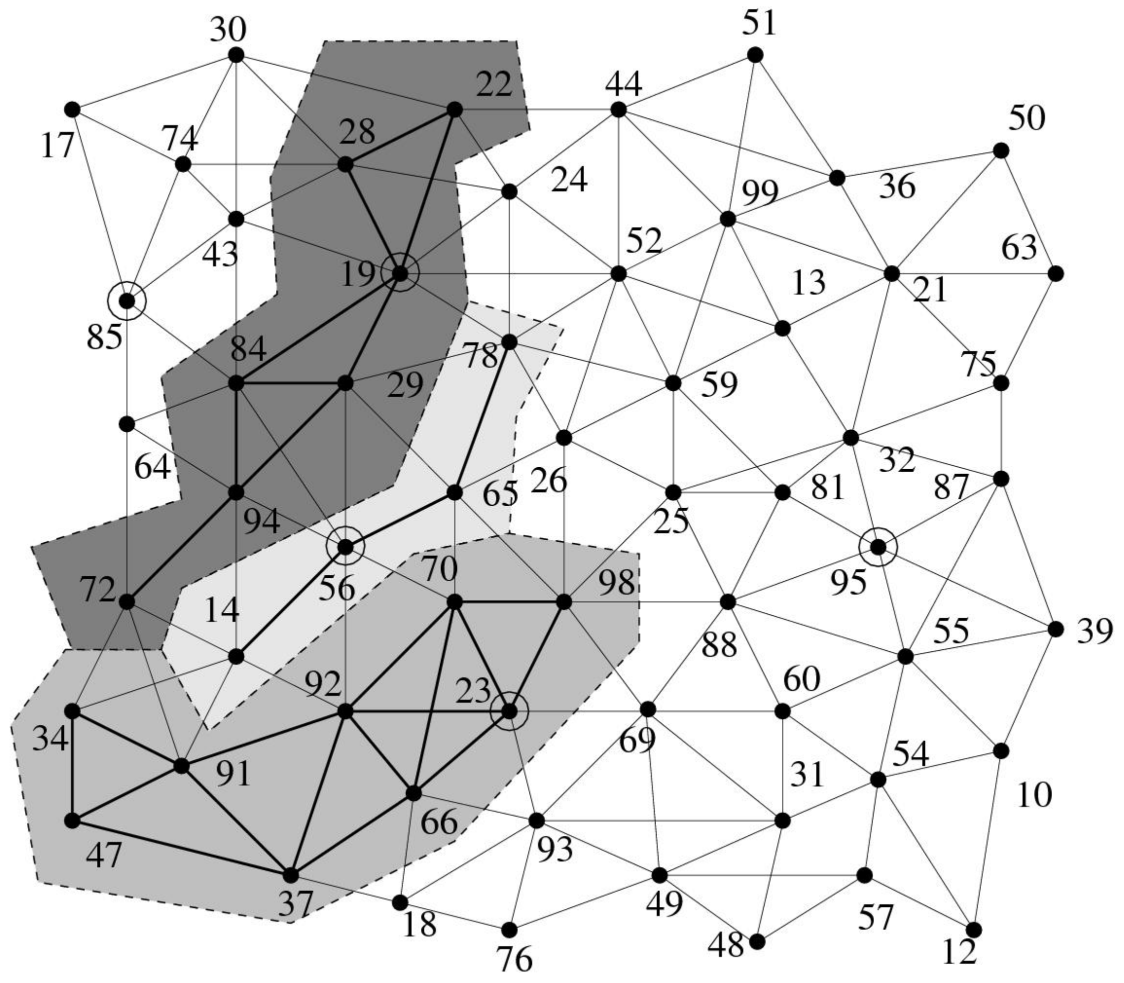


Figure 5.4 Temporary weak incompatibility
Weak incompatibility may not be permanent. Let *d_max* = 7. In this example, *G(56)* is compatible with both *G(19)* and *G(23)*, and *G(19)* and *G(23)* are weakly (but not strongly) incompatible. If *G(56)* merges with either of the others, the remaining two groups will be compatible, and can merge to include all the shaded area.

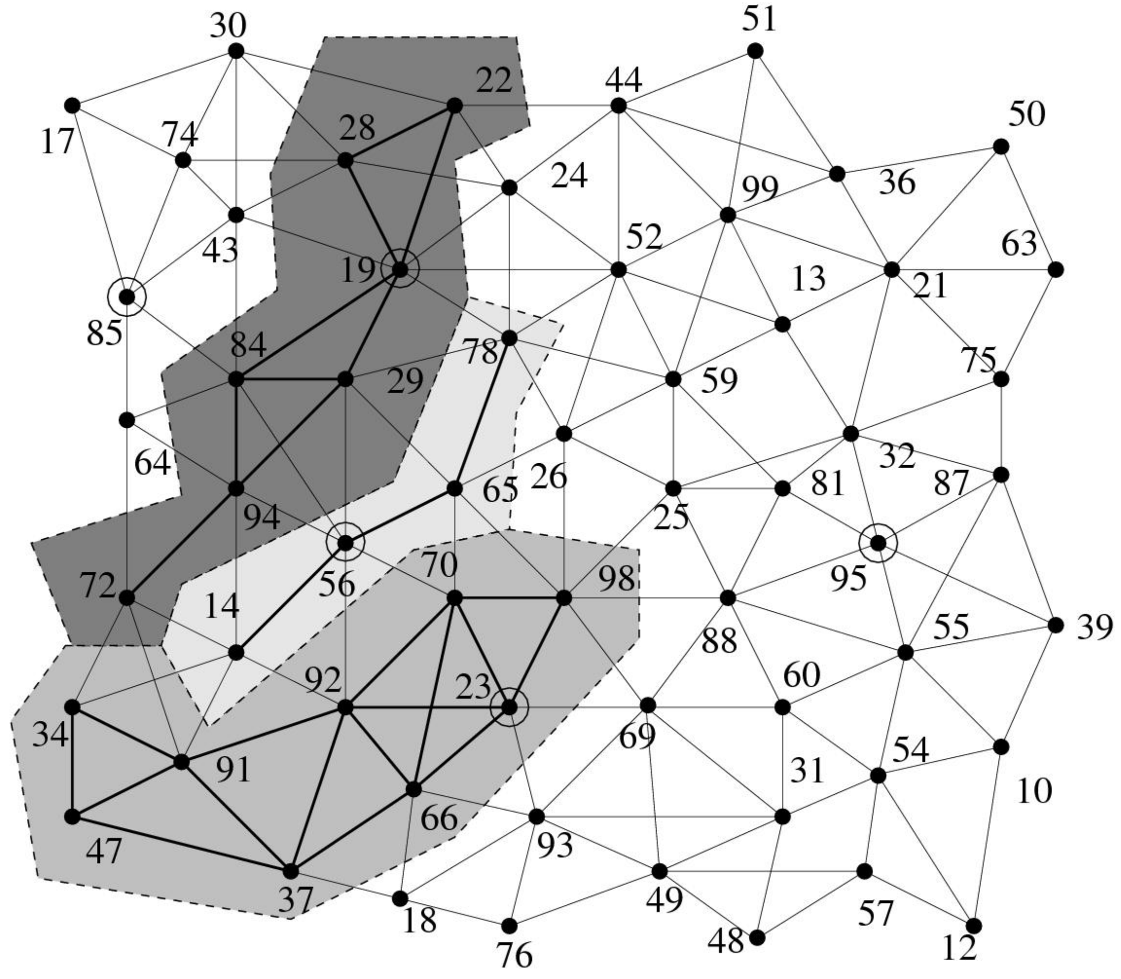


Figure 5.5 Weakly incompatibility deadlock
Let *d_max* = 7. If all three groups shown were combined, the resulting set would have diameter 7. However, any two of the three are weakly incompatible, so no merging can occur.

CHAPTER 6

PREPROCESSING

The preprocessing module is illustrated by a box in the diagram shown in Figure 5.1. Preprocessing consists of four sub-modules, which we now consider in detail.

6.1 Computation of *dist*[ ]

Comp(*dist*) is the submodule which computes $x.dist[\ ]$ for all x. For any given $y \in X$, the values $x.dist[y]$ for all $x \in X$ are computed by flooding, starting from y. After this computation converges, $x.dist[y] = d(x,y)$ if $d(x,y) \leq d\_max + 1$, and $x.dist[y] = \perp$ otherwise. Note that computation of the set of values $\{x.dist[z] : x \in X\}$, for $z \neq y$, are completely independent. Thus, all values of $x.dist[y]$ are computed using n independent algorithms running concurrently, one for each choice of y.

For any x and y, we define

$$Dist(x,y) = \begin{cases} 0 & \text{if } \ y = x \\ 1 + \min\{z.[y] : z \in N(x)\} & \text{if } \exists z \in N(x) : \ z.[y] \leq d\_max \\ \perp & otherwise \end{cases}$$

Action A1 of Table 6.1 then sets $x.dist[y] \leftarrow Dist(x,y)$.

6.2 Computation of the BFS and MIS Trees

We will assume the existence of a distributed algorithm, BFS-MIS, which elects a leader, *leader*_BFS , and constructs a BFS tree Γ_{BFS} of X rooted at *leader*_BFS. BFS-MIS also constructs a maximal independent set (MIS) of X, as well as a tree Γ_{MIS} also rooted at *leader*_BFS, which has the property that the MIS is the set of processes at even depth. We are not concerned about the details of BFS-MIS, but we require that it satisfies the following conditions.

1. BFS-MIS is self-stabilizing and silent.
2. Every process x has the following variables.

   (a) $x.level$_BFS the BFS *level* of x, the distance from x to *leader*_BFS.

   (b) $x.parent$_MIS, the parent of x in Γ_{MIS}.
3. MIS is a maximal independent set of processes of X. That is:

   (a) If $x, y \in$ MIS, then x and y are not neighbors.

   (b) If $x \notin$ MIS, then some neighbor of x is in MIS.
4. $x \in$ MIS if and only if the path in Γ_{MIS} from x to *leader*_BFS has even length.

Any algorithm which satisfies the specifications could be used, such as the algorithm given in [28]. Henceforth, we treat BFS-MIS as a "black box."

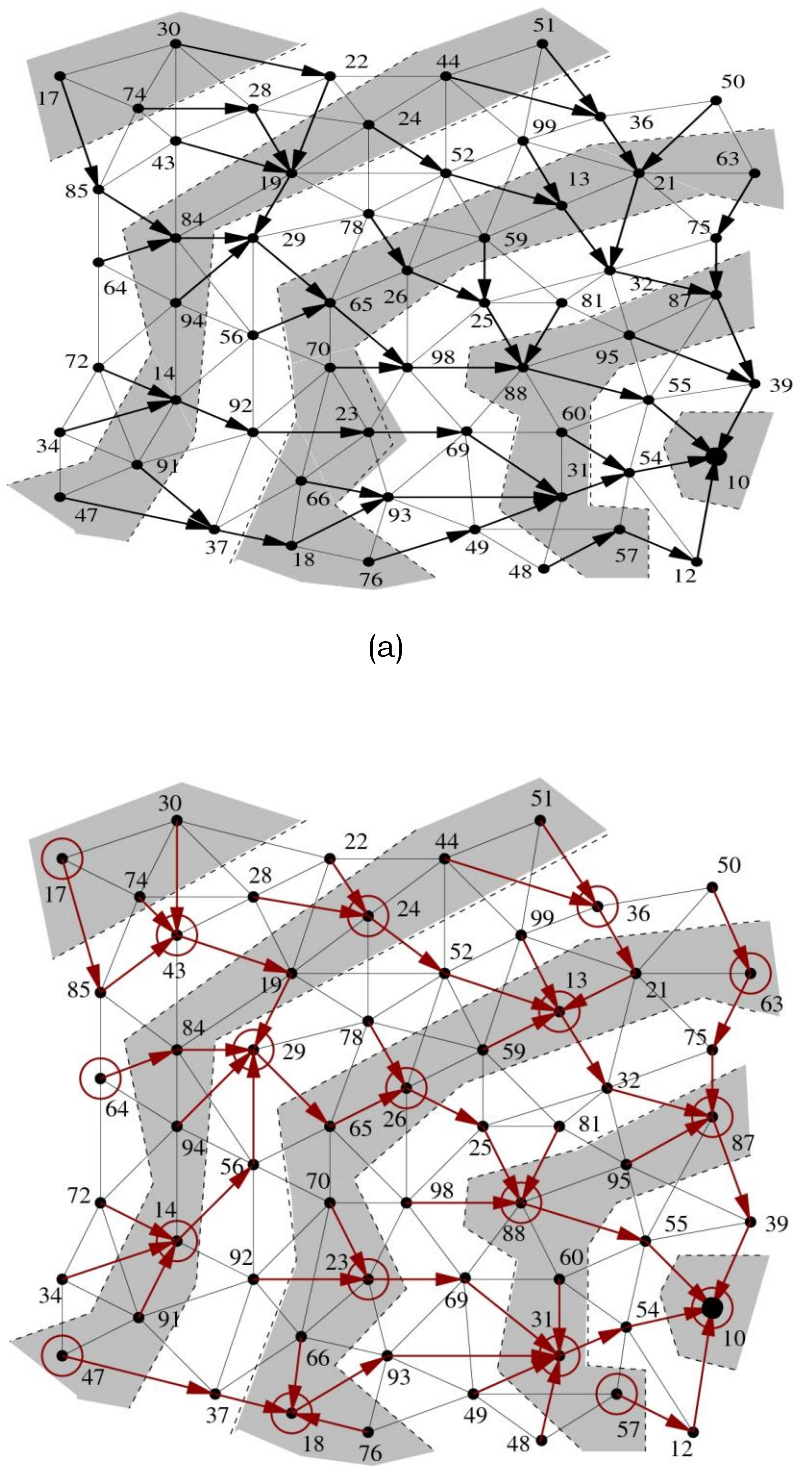


Figure 6.1 BFS tree (a) and MIS tree (b) of an example graph, constructed by BFS-MIS
Alternate BFS levels are shaded. In (b), members of MIS are circled.

6.3 Computation of β

The module BETA computes an integer $x.\beta \in \{0,1,\dots,\text{d\_max}\}$ for all x. The computation is bottom-up on Γ_{MIS}. We define $Beta(x)$ as a function of the values of $y.\beta$ for all children y of x, and then $x.\beta$ is set to $Beta(x)$. Before we give the formal definition of the correct values of $x.\beta$, we give the intuition behind that definition.

Our goal is to partition Γ_{MIS} into groups. Using β, we will construct a minimum partion of Γ_{MIS}, which we will call the initial partition of X. That is to say, if we delete all edges of X that are not edges of the tree Γ_{MIS}, no other partitions of Γ_{MIS} has fewer groups.

We first note that $x.\beta$ depends only on the topology of T_x, which we define to be the subtree of Γ_{MIS} rooted at x. We are actually constructing a partion of each T_x from the bottom up, using the following rules.

- The partition on T_x has as few groups as possible.
- The height of the top group of T_x, namely that group, which contains x, is as small as possible. The reason for this rule is that it allows the top group to capture as much of $\Gamma_{MIS} - T_x$ as possible. In fact, $x.\beta$ will be the height of that top group.

If x is a leaf, then T_x is a single point, and the partition of T_x consists of exactly one group which is a tree of height zero. Thus, $x.\beta = 0$. Otherwise, let $y_1, \dots, y_m$ be the children of x, and assume that partitions of all T_{y_i} have been constructed, and thus all $y_i.\beta$ are computed.

Consider the top groups of all T_{y_i}. Since we want to minimize the partition of T_x, we would like x to join together, into a single group, as many of the top groups of the subtrees as possible. If it is not possible to join two or more of those top groups into a single group, we would like x to join the subtree top group of smallest height, in order to allow maximum upward growth of the top group of T_x. If neither of those is possible, x will start a new group, *i.e.,* we let $x.\beta = 0$.

If the top group of any subtree T_{y_i} does not join with x, then y_i becomes the leader of one group of the initial partition. At the end of the construction, since there are no processes above *Root*_BFS, it must become the leader of its group.

We now give the formal definition of the function *Beta*. If x is a leaf of Γ_{MIS}, then $Beta(x) = 0$. Otherwise, $Beta(x)$ is as defined below.

1. If $\beta(y) = d\_max$ for all $y \in Children_{MIS}(x)$, then $Beta(x) = 0$. (Note that this covers the case where x is a leaf of Γ_{MIS}.)
2. Suppose $\beta(y) < d\_max$ for some $y \in Children_{MIS}(x)$.
   (a) Let $a = \min\{\beta(y) : y \in Children_{MIS}(x)\}$.
   (b) If $2a + 2 > d\_max$ then $Beta(x) = a + 1$.
   (c) If $2a + 2 \leq d\_max$, let $b = \max\{\beta(y) : y \in Children_{MIS}(x)$ and $\beta(y) + a + 2 \leq d\_max\}$.
   (Note that $a \leq b \leq d\_max - a - 2$.) Then $Beta(x) = b + 1$.

Action A3 of Table 7.1 sets $x.\beta \leftarrow Beta(x)$

6.4 The Initial Partition: Computation of *init_leader*

Once β is defined, we construct the initial partition, which is the minimum partition of Γ_{MIS}, by deleting some of the edges of Γ_{MIS}. Each resulting component will be a group of the initial partition. The rules for deletion of edges are given below.

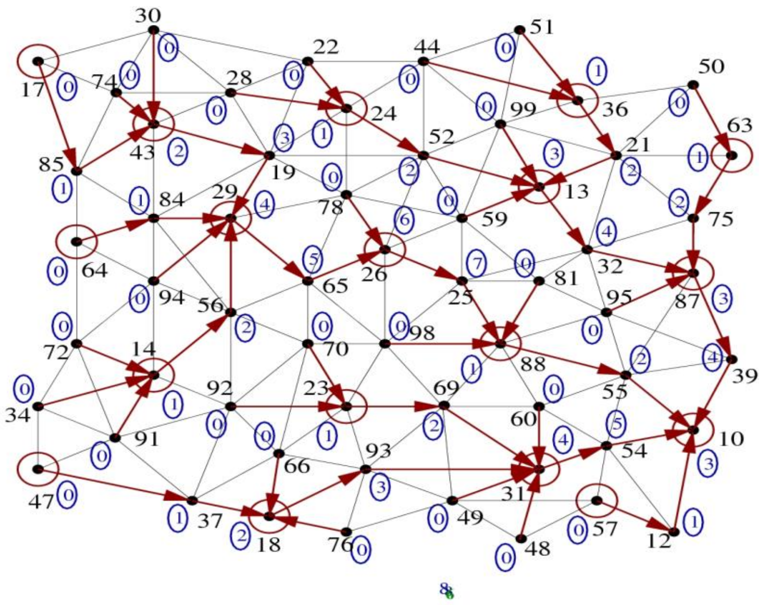
Suppose x is a process which is not a leaf of Γ_{MIS}, and $\{y_1, \ldots y_k\}$ is the set of children of x in Γ_{MIS}. We will delete the edge from y_i to x if and only if the top group of T_{x} does not include y_i. We renumber the children so that $\beta(\text{y}_1) + 1 = \beta(\text{x})$.

- If $\beta(\text{x}) \leq d\_max/2$, then we delete the edge $\{\text{y}_i, \text{x}\}$ if and only if $\beta(\text{y}_i) \geq \beta(\text{x})$.
- If $\beta(x) > d\_max/2$, then we delete the edge $\{\text{y}_\text{i}, \text{x}\}$ if and only if $i > 1$ and $\beta(y_i) + \beta(y_1) + 2 > d\_max$.

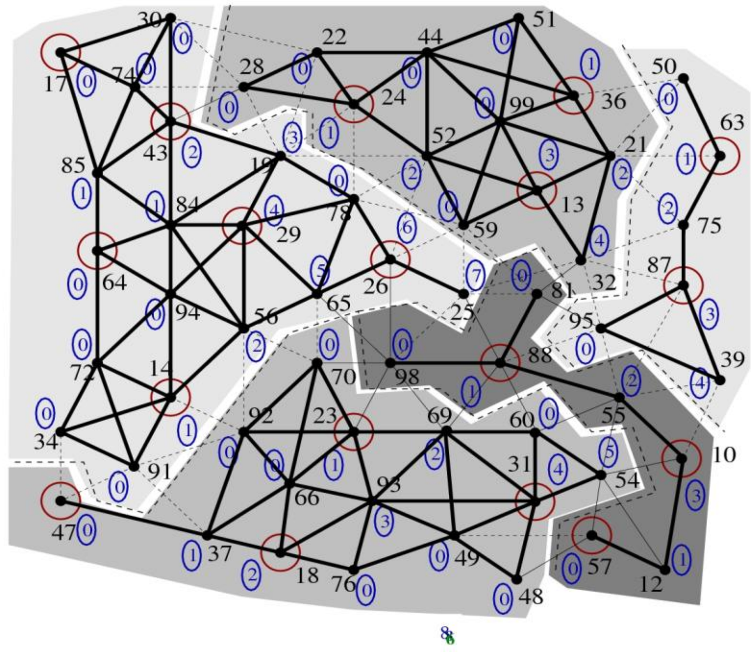
The resulting graph, after deleting those edges from Γ_{MIS}, consists of the union of components, $T_1, \ldots, T_m$, which are trees. Each of these components T_i then defines a group G_i, defined to be the full subgraph of X whose processes are the same as those of T_i. We let the leader of each group be the highest process in the group, *i.e.*, the process closest to *Root*_BFS.

Using the above rules, we can define a function on process as follows:

$$Init\_Leader(\text{x}) = \begin{cases} x & \text{if } x = Root\_\text{BFS or } \{x, x.parent\_\text{MIS}\} \text{ is deleted} \\ x.parent\_\text{MIS}.init\_leader & \text{otherwise} \end{cases}$$



(a)



(b)

Figure 6.2 (a) the function β for the example network, where $d\_max = 7$, and (b) the resulting initial partition.

Lemma 6.1

(a) For any i, $(G_i) \leq d\_max$.

(b) All but possibly one G_i contains at least $\lfloor (d\_max + 2)/4 \rfloor$ members of the *MIS*.

Finally, the code for the entire preprocessing phase is given in Table 6.1 below. Using the same notation as earlier, let BFS-MIS-*Enabled*(x) be the predicates such that x is enabled to execute an action of BFS-MIS. Action A6 in the table is necessary to satisfy Specification 3a given in Section 5.3. This is necessary to permit the first execution of Front to proceed, in case of erroneous initialization of the variable *x.weak_cert*[ ] for some *x*. This issue will be discussed in detail in Section 10.

Table 6.1: Actions of Module PREPROCESS

Label	Name	Guard	Statement
A1 Priority 1	DIST	$x.dist[y] \neq Dist(x,y)$	$\longrightarrow x.dist[y] \leftarrow Dist(x,y)$
A2 Priority 2	BFS-MIS	$\text{BFS} - \text{MIS} - Enabled(x)$	$\longrightarrow x$ executes an action of BFS-MIS
A3 Priority 3	Beta	$x.\beta \neq Beta(x)$	$\longrightarrow x.\beta \leftarrow Beta(x)$
A4 Priority 4	Init Leader	$x.init\_leader \neq Init\_Leader(x)$	$\longrightarrow x.init\_leader \leftarrow Init\_Leader(x)$
A5 Priority 5	Leader	$x.leader \neq x.init\_leader$	$\longrightarrow x.leader \leftarrow x.init\_leaderader(x)$
A6 Priority 6	Clear Weak Certificate	$x.weak\_cert[\ell] \neq \perp$	$x.weak\_cert[\ell] \leftarrow \perp$

CHAPTER 7

FRONT MODULE

We will refine the flow diagram slightly, by adding two submodules to Front. The module Front, illustrated by the second large box in Figure 7.1, is the concatenation of nine submodules, which we now describe in detail. The variables x.*leader* and x.*weak_cert*[ ] are never changed during an execution of Front.

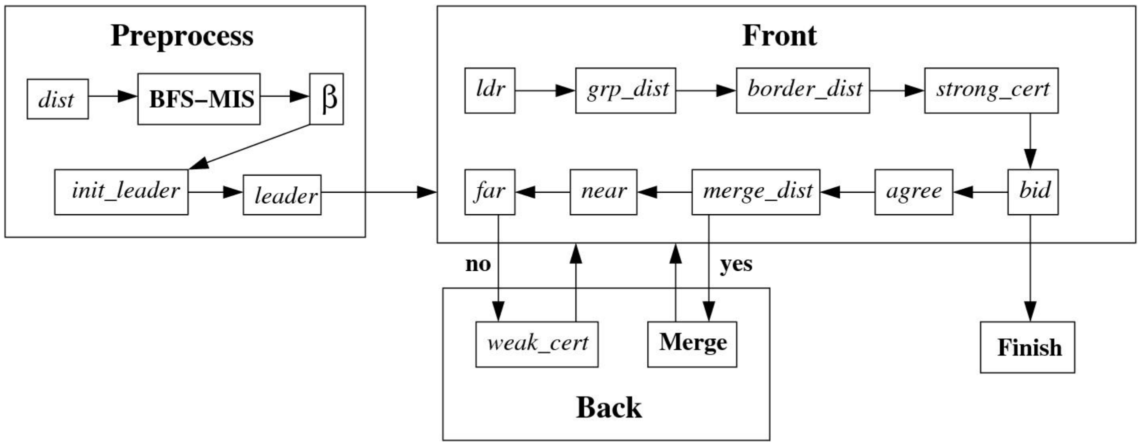


Figure 7.1 Normal flow of the algorithm.
The boxes indicate individual modules.

7.1 Computation of *ldr*[ ]

The first box inside the module Front in Figure 7.1 represents the submodule that computes the dynamic array x.*ldr*[ ], for all x. When that computation converges, $x.ldr[y] = y.$ for all $y \in H_{d\_max+1}(x)$. The dual version of that statement is that, for each given y, $x.ldr[y] = y.ldr$ for all

$x \in H_{d\_max+1}(y)$. The dual version gives better intuition for the calculation, which is by a top-down wave starting at $y.ldr[y]$, which is set to $y.leader$.

During subsequent executions of Front, the value of $x.ldr[y]$ will change if $y.leader$ has changed. We define:

$$Z(x,y) = \{z \in N(x) : z.dist[y] + 1 = x.dist[y]\}$$

$$Ldr(x,y) = \begin{cases} x.leader & \text{if } y = x \\ \min\{z.ldr[y] : z \in Z(x,y)\} & \text{if } Z(x,y) \neq \emptyset \\ \perp & \text{otherwise} \end{cases}$$

Action A1 of Table 7.1 then sets $x.ldr[y] \leftarrow Ldr(x,y)$.

7.2 Computation of *grp_dist*[ ]

For each process x, $x.grp\_dist[\ ]$ is a dynamic array. The correct range of $x.grp\_dist[\ ]$ is $G(x)$, and the correct value of $x.grp\_dist[y]$ is $d_{G(x)}(x,y)$ for all $y \in G(x)$.

This array is used for error checking. If $x.grp\_dist[y]$ does not converge to an integer in the range $\{0 \ldots d\_max\}$ for all y such that $x.ldr[y] = x.ldr[x]$, then x has detected an error.

We define:

$$NG(x,y) = \{z \in N(x) : x.ldr[z] = x.ldr[x] \text{ and } z.grp\_dist[y] \leq d\_max\}$$

$$Grp\_Dist(x,y) = \begin{cases} 1 + \min\{z.grp\_dist[y] : z \in NG(x,y)\} & \text{if } NG(x,y) \neq \emptyset \\ \perp & \text{otherwise} \end{cases}$$

Action A2 of Table 7.1 then sets $x.grp\_dist[y] \leftarrow Grp\_Dist(x,y)$.

7.3 Computation of *border_dist*[ ]

For each process x, after the dynamic array $x.border\_dist[\ ]$ converges its range will be the set of leaders of all groups which neighbor $G(x)$. The purpose of this array is for each process to know the neighbor groups of its group. The array converges by simple flooding, starting by assigning $x.border\_dist[\ell]$ to zero if x does not belong to $\mathrm{G}(\ell)$ and is adjacent to a process which belongs to $\mathrm{G}(\ell)$. The correct value of $x.border\_dist[\ell]$ is the shortest length of any path in $G(x)$ from x to some process of $G(x)$ which borders $\mathrm{G}(\ell)$.

We define:

$$NB(\mathrm{x},\ell) = \{\, z \in N(x) : x.ldr[z] = x.ldr[x] \text{ and } z.border\_dist[\ell] \le d\_max \,\}$$

$$Border\_Dist(\mathrm{x},\ell) = \begin{cases} 0 \quad \text{if } x.ldr[\mathrm{x}] \ne \ell \text{ and } N(x) \cap \mathrm{G}(\ell) \ne \emptyset \\ 1 + \min\{\mathrm{z}.border\_dist[\ell] : \mathrm{z} \in NB(\mathrm{x},\ell)\} \ \text{ if } NB(\mathrm{x},\ell) \ne \emptyset \\ \perp \quad\ \ \text{otherwise} \end{cases}$$

Action A3 of Table 7.1 then sets $x.border\_dist[\ell] \leftarrow Border\_Dist(x,\ell)$.

Lemma 7.1 If *COMP*(ldr), *COMP*(grp_dist), and *COMP*($border\_dist$) have converged, then $\mathrm{x}.border\_dist[\ell]$ is defined if and only if ℓ is the leader of a group which is adjacent to $G(x)$.

7.4 Computing *strong_cert*[ ]

The most difficult part of the algorithm is deciding whether to merge two neighboring groups. Suppose that m and ℓ are leaders of neighboring groups, and that $\mathrm{m} < l$. The groups G(m) and $\mathrm{G}(\ell)$ can be merged if and only if G(m) and $\mathrm{G}(\ell)$ are compatible, *i.e.,* $diam(\mathrm{G(m)} \cup \mathrm{G}(\ell)) \le d\_max$. Thus, G(m) and $\mathrm{G}(\ell)$ are incompatible if and only if

$$\exists x \in G(m)\ \exists y \in G(\ell) \text{ such that } d_{G(m)\cup G(\ell)}(x,y) > d\_max$$

Since the groups are adjacent and both groups are connected, we can simplify the condition: $\mathrm{G(m)}$ and $\mathrm{G}(\ell)$ are incompatible if and only if

$\exists \mathrm{x} \in \mathrm{G(m)}\ \exists \mathrm{y} \in \mathrm{G}(\ell)\ \mathrm{d}_{\mathrm{G(m)} \cup \mathrm{G}(\ell)}(\mathrm{x}, \mathrm{y}) = d\_max + 1$

Recall that $\mathrm{G(m)}$ and $\mathrm{G}(\ell)$ are strongly incompatible if $\mathrm{d(x, y)} = d\_max + 1$ for some $\mathrm{x} \in \mathrm{G(m)}$ and some $\mathrm{y} \in \mathrm{G}(\ell)$. Strong incompatibility implies incompatibility, since $\mathrm{d(x, y)} \le \mathrm{d}_{\mathrm{G(m)} \cup \mathrm{G}(\ell)}(\mathrm{x, y})$.

The purpose of the array $\mathrm{x}.strong\_cert[\ ]$ is to certify strong incompatibility. In fact, after stabilization of Front, $\mathrm{G(m)}$ is strongly incompatible with $\mathrm{G}(\ell)$ if and only if $\mathrm{x}.strong\_cert[\ell] \neq \perp$ for some $\mathrm{x} \in \mathrm{G(m)}$, which in turn implies that $\mathrm{x}.strong\_cert[\ell] \neq \perp$ for all $\mathrm{x} \in \mathrm{G(m)}$.

Let x be a process. Suppose ℓ is the leader of a group which is a neighbor of $\mathrm{G(x)}$. If $\mathrm{G}(\ell)$ is strongly incompatible with $\mathrm{G(x)}$, the correct value of $\mathrm{x}.strong\_cert[\ell]$ is the shortest distance, through $\mathrm{G(x)}$, to some $\mathrm{y} \in \mathrm{G(x)}$ such that $\mathrm{d(y, v)} = d\_max + 1$ for some $\mathrm{v} \in \mathrm{G}(\ell)$; formally stated:

$\min\{\mathrm{d}_{\mathrm{G(x)}}(\mathrm{x, y}) : \mathrm{y} \in \mathrm{G(x)} \text{and}\ \exists \mathrm{v} \in \mathrm{G}(\ell) : \mathrm{d(y, v)} = d\_max + 1\}$

Note that if $\mathrm{G(x)}$ and $\mathrm{G}(\ell)$ are not strongly incompatible, the above formula is undefined.

The values of $strong\_cert$ are computed recursively. For any x and any ℓ, we define:

$NS(\mathrm{x}, \ell) = \{\mathrm{z} \in \mathrm{N(x)} : \mathrm{x}.ldr[\mathrm{z}] = \mathrm{x}.ldr[\mathrm{x}] \text{and}\ \mathrm{z}.strong\_cert[\ell] < d\_max\}$

$$Strong\_cert(\mathrm{x}, \ell) = \begin{cases} 0 & \text{if } \exists \mathrm{v} : \mathrm{x}.dist[\mathrm{v}] = d_{max} + 1 \text{ and } \mathrm{x}.ldr[w] = \ell \\ 1 + \min\{\mathrm{z}.strong\_cert[\ell] : \mathrm{z} \in NS(\mathrm{x}, \ell) & \text{if } NS(\mathrm{x}, \ell) \neq \emptyset \\ \perp & \text{otherwise} \end{cases}$$

Action A4 of Table 7.1 then sets $x.strong\_cert[\ell] \leftarrow Strong\_cert(x, \ell)$.

After the values of the dynamic array $x.strong\_cert[\ ]$ stabilize for all x, a non-null value of $x.strong\_cert[\ell]$ certifies that G(x) and G(ℓ) are strongly incompatible, and hence cannot merge.

Suppose m and ℓ are leaders of two neighboring groups. After stabilization of COMP(*strong_cert*), as well as the three earlier submodules of FRONT, one of two situations holds.

1. If $d(x, y) \leq d\_max$ for all $x, y \in G(m) \cup G(\ell)$, then $x.strong\_cert[\ell] = \perp$ for all $x \in G(m)$ and $y.strong\_cert[m] = \perp$ for all $y \in G(\ell)$.
2. Otherwise, $x.strong\_cert[\ell] \neq \perp$ for all $x \in G(m)$ and $y.strong\_cert[m] \neq \perp$ for all $y \in G(\ell)$. For a given $x \in G(m)$, there must exist some $y \in G(\ell)$ and $z \in G(m)$ such that $d(y, z) = d\_max + 1$, and the correct value of $x.strong\_cert[\ell]$ is the shortest distance to such a choice of z. More formally:

$$x.strong\_cert[\ell] = \min\{d(x, z) : z \in G(m) \wedge$$
$$(\exists y \in G(\ell) : d(z, y) = d\_max + 1)\}$$

In this situation, ℓ and m will never be able to be part of the same group.

7.5 Computation of *bid*, *agree*, and *merge_dist*

After *strong_cert* has been correctly computed, each group decides to attempt to merge with a neighboring group, provided there exists a neighboring group which might still be compatible. Each process x computes $x.bid$, which is the leader of the neighboring group that x has "bid" to merge with. (If all groups which neighbor $G(x)$ are already known by x to be incompatible, then $x.bid \leftarrow \perp$.) The bid is uniform, *i.e.,* if $z \in G(x)$, then $z.bid = x.bid$.

The variable $x.agree$ is Boolean. Write $m = x.ldr[x]$. If, after *bid* has stabilized, $x.bid = \ell$, where ℓ is the leader of a neighboring group, and $y.bid = m$ for all $y \in G(\ell)$, then there is an agreement to attempt to merge between $G(m)$ and $G(\ell)$. In this case, $x.agree$ and $y.agree$ will both be computed to be true for all $y \in G(m) \cup G(\ell)$. On the other hand, if $x.bid = \ell$ and $y.bid \neq m$ for all $y \in G(\ell)$, then $x.agree$ will be computed to be false.

After *agree* has stabilized, $x.merge\_dist[\ ]$ will be computed for all x. If $x.agree$ is false, then $x.merge\_dist[\ ] \leftarrow \perp$ for all y. On the other hand, suppose $x.ldr[x] = m$ and $x.bid = \ell$, as before; and $x.agree$ is true. Then the correct value of $x.merge\_dist[y]$ is $d_{G(m) \cup G(\ell)}(x, y)$ for all $y \in G(m) \cup G(\ell)$. After $merge\_dist$ has stabilized, $G(m)$ and $G(\ell)$ are compatible if and only if $x.merge\_dist[y] \leq d\_max$ for all $x, y \in G(m) \cup G(\ell)$.

We now show how our algorithm computes these variables. It is necessary to know the values of $x.weak\_cert[\ ]$ to make these computations, values which were computed during previous iterations of Module Back. If $x.weak\_cert[\ell] \neq \perp$, and the values of $weak\_cert$ are correct, then $G(m)$ and $G(\ell)$ are weakly incompatible. We will explain the structure and computation of $weak\_cert$ in Section 8.1.

Define a Boolean function $Poss\_Comp(x, \ell)$, for x and ℓ, meaning that $G(x)$ and $G(\ell)$ are "possibly compatible," as follows.

$$Poss\_Comp(x, \ell) = \begin{cases} \text{TRUE} & \text{if } x.border\_dist[\ell] \neq \perp \text{ and } \\ & \quad x.strong\_cert[\ell] = \perp \text{ and } \\ & \quad x.weak\_cert[\ell] = \perp \\ \text{FALSE} & \text{otherwise} \end{cases}$$

For any process x, we define:

$$Poss\_Comp\_Grps(x) = \{ \ \ell : Poss\_Comp(x, \ell) \ \}$$

$$Bid(x) = \begin{cases} \min Poss\_Comp\_Grps(x) & \text{if } Poss\_Comp\_Grps(x) \neq \emptyset \\ \perp & \text{otherwise} \end{cases}$$

For any process x and for $\ell = x.bid$, we then define:

$Agree(x)$

$$= \begin{cases} \text{TRUE} & \text{if } \exists \text{y} \in N(x) : \text{y}.ldr[\text{y}] = \ell \text{ and y}.bid = x.ldr[x] \\ \text{TRUE} & \text{if } \exists z \in NG(x) : z.agree \text{ and z}.border\_dist[\ell] + 1 = x.border\_dist[\ell] \\ \text{FALSE} & \text{otherwise} \end{cases}$$

$$NY(x) = \{ \ \text{y} : x.ldr[\text{y}] \in \{\ell, x.ldr[x] \} \}$$

$$NZ(x) = \text{N}(x) \cap NY(x)$$

$$NL(x, y) = \{ z \in NZ(x) : z.merge\_dist[\text{y}] \leq 2 \ d\_max \}$$

$Merge\_Dist(x, y)$

$$= \begin{cases} 0 & \text{if } x.agree \text{ and } y = x \\ 1 + \min \{ \text{z}.merge\_dist[\text{y}] : \text{z} \in NL(x, y) & \text{if x}.agree \text{ and y} \neq x \\ \perp & \text{otherwise} \end{cases}$$

Action A5 of Table 7.1 then sets $x.bid \leftarrow Bid(x)$, Action A6 sets $x.agree \leftarrow Agree(x)$, and Action A7 sets $x.merge\_dist[y] \leftarrow Merge\_Dist(x, y)$.

Lemma 7.2 If $x.bid = \ell$, all previous submodules of Front have converged, and there are no errors, then $NY(\mathrm{x}) = \mathrm{G}(\mathrm{x}) \cup \mathrm{G}(\ell)$, and $x.merge\_dist[y] = d_{NY(x)(}x, y)$ for all $y \in (x)$.

7.6 Computation of *near* and *far*

We now assume that the first seven submodules of Front have stabilized. The value of $x.near$ is computed for each process x. After the computation of $near$ has stabilized, $x.far$ is computed for each process x. If $x.agree$ is false, then $x.near$ and $x.far$ will be computed to be $\perp$.

On the other hand, consider two neighboring groups with leaders m and ℓ. Without loss of generality, $m < l$. Suppose $m.bid = \ell$ and $\ell.bid = m$. Then, $x.agree$ is true for all $x \in G(m) \cup G(\ell)$. For all $x \in G(m) \cup G(\ell)$, we will compute $x.near$ to be the minimum $u \in G(m)$ whose distance from some process in $G(\ell)$ is $d\_max + 1$, and we will compute $x.far$ to be the minimum $v \in G(\ell)$ such that $d_{G(m) \cup G(\ell)} = d\_max + 1$.

We define the following functions.

$$Bst\_Nbr\_Near(x) = \min\{\mathrm{z}.near : \mathrm{z} \in NZ(x) \text{ and } \mathrm{z}.merge\_dist[\mathrm{z}.near] + 1 = x.merge\_dist[\mathrm{z}.near]\}$$

$$Near(x) = \begin{cases} \min\{x, Bst\_Nbr\_Near(x)\} & \text{if } x.ldr[x] = \ell \text{ and } \exists \mathrm{v} : x.merge\_dist[\mathrm{v}] = d\_max + 1 \\ Bst\_Nbr\_Near(x) & \text{otherwise} \end{cases}$$

$$Far(x) = \begin{cases} \min\{\mathrm{v} : x.merge\_dist[\mathrm{v}] = d\_max + 1\} & \text{if } x.near = x \\ \min\{z.far : \mathrm{z} \in \mathrm{NZ}(x) \text{ and } z.merge\_dist[x.near] + 1 = x.merge\_dist[x.near]\} & \text{otherwise} \end{cases}$$

Action A8 of Table 7.1 then sets $x.near \leftarrow Near(x)$, and Action A9 sets $x.far \leftarrow Far(x)$

Lemma 7.3 If Front has converged, and if m $< l$ are leaders of adjacent groups such that m.$bid = \ell$ and $\ell.bid =$ m, then:

(a) If $diam\big(G(m) \cup G(\ell)\big) \leq d\_max$, then $x.near = x.far = \perp$ for all $x \in G(m) \cup G(\ell)$.

(b) If $diam\big(G(m) \cup G(\ell)\big) \geq d\_max + 1$, then there exist process $u \in G(m)$ and $v \in G(\ell)$ such that

(i) $x.near = u$ and $x.far = v$ for all $x \in G(m) \cup G(\ell)$.

(ii) $d_{G(m) \cup G(\ell)}(u, v) = d\_max + 1$.

Table 7.1: Module Front for Process x

Label	Name	Guard		Statement
A1 Priority 1	Ldr	$\mathrm{x}.ldr[\mathrm{y}] \neq Ldr(\mathrm{x},\mathrm{y})$	$\longrightarrow$	$\mathrm{x}.ldr[\mathrm{y}] \leftarrow Ldr(\mathrm{x},\mathrm{y})$
A2 Priority 2	Group Dist	$x.grp\_dist[y] \neq Grp\_Dist(x,y)$	$\longrightarrow$	$x.grp\_dist[y] \leftarrow Grp\_Dist(x,y)$
A3 Priority 3	Border Dist	$x.border\_dist[y] \neq Border\_Dist(x,y)$	$\longrightarrow$	$x.border\_dist[y] \leftarrow Border\_Dist(x,y)$
A4 Priority 4	Strong Certificate	$x.stron\_cert[y] \neq Strong\_Cert(x,y)$	$\longrightarrow$	$x.stron\_cert[y] \leftarrow Strong\_Cert(x,y)$
A5 Priority 5	Bid	$\mathrm{x}.bid \neq Bid(\mathrm{x})$	$\longrightarrow$	$\mathrm{x}.bid \leftarrow Bid(\mathrm{x})$
A6 Priority 6	Agree	$x.Agree \neq Agree(x)$	$\longrightarrow$	$x.Agree \leftarrow Agree(x)$
A7 Priority 7	Merge Dist	$x.merge\_dist \neq Merge\_Dist(x)$	$\longrightarrow$	$x.merge\_dist \leftarrow Merge\_Dist(x)$
A8 Priority 8	Near	$x.near \neq Near(x)$	$\longrightarrow$	$x.near \leftarrow Near(x)$
A9 Priority 9	Far	$x.far \neq Far(x)$	$\longrightarrow$	$x.far \leftarrow Far(x)$

CHAPTER 8

BACK MODULE

We now give a detailed description of the module Back, which consists of two submodules, Merge and Comp($weak\_cert$). Suppose $x.ldr[x] = m$, $x.bid = \ell$, and $x.agree = \text{TRUE}$. If $x.near = \perp$, then $G(m)$ and $G(\ell)$ will merge during the execution of Back by executing the submodule Merge. If, on the other hand, $x.near \neq \perp$, $G(m)$ and $G(\ell)$ will not merge; instead, all of $G(m) \cup G(\ell)$ will construct a weak certificate by executing the submodule Comp($weak\_cert$). This weak certificate will remain in place until either $G(m)$ or $G(\ell)$ merges with another group.

The submodule Merge has another task, namely to delete out-of-date weak certificates. Suppose $G(m)$ and $G(\ell)$ merge. Then all previously existing weak certificates which involve either $G(m)$ or $G(\ell)$ must be deleted.

8.1 Computation of $weak\_cert$[ ]

A weak certificate is a 4-tuple of variables: $x.weak\_cert[\ell] = (x.u[\ell]$, $x.d_u[l]$, $x.v[\ell]$, $x.d_v[\ell])$. For short, we will let $\perp$ also denote the 4-tuple $(\perp, \perp, \perp, \perp)$.

We define the function $Weak\_Cert(x) = (x.near, x.merge\_dist[x.near]$, $x.far$, $merge\_dist[x.far]))$. If the configuration is not erroneous, and if $x.agree$ is true and $x.near = \perp$, or if $x.agree$ is false, then all the component functions of $Weak\_Cert(x)$ are undefined, in which case we can say $Weak\_Cert(x) = \perp$.

Action A1 of Table 8.1 then sets $x.weak\_cert[\ell] \leftarrow Weak\_Cert(x)$, provided $\ell = x.bid$.

We now give the intuition for weak certificates. Suppose $x.bid = \ell$ and $x.ldr[x] = m$, and the configuration is not in error. If $Weak\_Cert(x) \neq \perp$, that means that $G(m)$ and $G(\ell)$ are weakly incompatible.

Weak incompatibility of two groups $G(m)$ and $G(\ell)$ is discovered by examining the dynamic arrays $x.merge\_dist[\ ]$ for all $x \in G(m) \cup G(\ell)$. The size of each such dynamic array is the cardinality of $G(m) \cup G(\ell)$, which is within the allowed space complexity of our algorithm. However, if, as the algorithm proceeds, each process must store that array for each neighboring group with which its group is weakly incompatible, and given that the number of such groups is $O(n/d\_max)$, the total memory required for such storage is $O(n^2/d\_max)$. This could exceed our allowed space bound of $O(n)$ per process.

The weak certificates solve this problem by certifying weak incompatibility using much less space. For each ℓ, $x.weak\_cert[\ell]$ has space complexity $O(1)$. Thus, even if $x.weak\_cert[\ell]$ is defined for every possible ℓ, the space requirement for each to store all needed weak certificates is $O(n/d\_max)$.

8.2 Merge

To implement the submodule Merge, we define three functions.

$Merge(x) = x.agree\ \ and\ x.far = \perp$

$$Leader(x) = \begin{cases} x.bid & \text{if } Merge(x) \text{ and } x.bid < x.ldr[x] \\ x.ldr[x] & \text{otherwise} \end{cases}$$

$$Safe\_Weak\_Cert(x,\ell) = \forall y \in N(x)\ (\,y.ldr[y] = x.ldr[x]) \Rightarrow y.weak\_cert[\ell] \neq \perp$$
$$\text{and } \forall y \in N(x)\ (\,y.ldr[y] = \ell\,) \Rightarrow y.weak\_cert[x.ldr[x]] \neq \perp$$

$Merge(x)$ is true if x lies in a group that must be merged with another group. $Leader(x)$ is the leader of x after merging takes place. If $x.weak\_cert[\ell] \neq \perp$, then $Safe\_Weak\_Cert(x,\ell)$ indicates that the neighbors of x have corresponding certificates if they are either in $G(x)$ or $G(\ell)$. If $x.weak\_cert[\ell] \neq \perp$ and $Safe\_Weak\_Cert(x,\ell)$ does not hold, or if $Merge(x)$ holds, then $x.weak\_cert[\ell]$ is part of an out-of-date weak certificate, and must be deleted.

Table 8.1: Module Back for Process x

Label	Name	Guard	Statement
A1	Weak Certificate	$x.bid \neq \perp$ $x.weak\_cert[x.bid] \neq Weak\_Cert(x)$	$\longrightarrow x.weak\_cert[x.bid] \leftarrow Weak\_Cert(x)$
A2	Delete Weak Certificate	$x.weak\_cert[\ell] \neq \perp$ $Merge(x)$ or $\neg Safe\_Weak\_Cert(x,\ell)$	$\longrightarrow x.weak\_cert[\ell] \leftarrow \perp$
A3	Merge	$x.leader \neq Leader(x)$ $\forall y \in N(x)$ $:\ y.weak\_cert[x.leader] = \perp$	$\longrightarrow x.leader \leftarrow Leader(x)$

CHAPTER 9

ERROR DETECTION

We have defined our algorithm to be Preprocess+LOOP(Front,Back), using the construction given in Section 4.3. To apply the construction, we let $\mathcal{P}$ = Preprocess, $\mathcal{A}$ = Front, $\mathcal{B}$ = Back. We also define functions *Front_Ok* and *Back_Ok*, which play the role of the predicates $\mathcal{A}$*_Ok* and $\mathcal{B}$*_Ok*, respectively. These predicates must be defined so as to satisfy the list of specifications given in Section 4.3.

The sets of configurations in Figure 4.3 can then be defined as follows for our application:

- $\mathbb{C}$ is the set of all configurations.
- $\mathbb{P}$ is the set of all configurations where Preprocess is silent.
- $\mathbb{D}$ is the set of all configurations where $Front\_Ok(x)$ holds for each process x.
- $\mathbb{E}$ is the set of all configurations where $Back\_Ok(x)$ holds for each process x.
- $\mathbb{A}$ is the set of all configurations where $Front\_Ok(x)$ holds for each process x, and no process is enabled to execute an action of Front.
- $\mathbb{B}$ is the set of all configurations where $Back\_Ok(x)$ holds for each process x, and no process is enabled to execute an action of Back.
- $\mathbb{A} \cap \mathbb{B}$ is the set of legitimate configurations of our algorithm.

We define the following predicates for each process x. Each of these predicates means that a specific variable appears, to x, to have the correct value.

- $Dist\_Ok(x) \equiv x.dist[y] = Dist(x, y)$ for all y.
- BFS-MIS_$Ok(x) \equiv \neg$ BFS-MIS-$Enabled(x)$.
- $Beta\_Ok(x) \equiv x.\beta = Beta(x)$.
- $Leader\_init\_Ok(x) \equiv x.init\_leader = Leader\_init(x)$.
- $Leader\_Ok(x) \equiv \forall y \in N(x) : (y.leader = x.leader) \Rightarrow (y.init\_leader = x.init\_leader)$.
- $Ldr\_Ok(x) \equiv \forall y \neq x : x.ldr[y] = Ldr(x, y)$

  Note that we require that $y \neq x$ in this definition. The reason is that, otherwise, we would require that $x.ldr[x] = x.leader$. This condition is not maintained during the execution of Back, and hence would result in the entire algorithm starting over every time Back executes.
- $Grp\_Dist\_Ok(x) \equiv \forall y : x.grp\_dist[y] = Grp\_Dist(x, y)$
- $Border\_Dist\_Ok(x) \equiv \forall \ell : x.border\_dist[\ell] = Border\_Dist(x, \ell)$
- $Strong\_Cert\_Ok(x) \equiv \forall \ell : x.strong\_cert[\ell] = Strong\_Cert(x, \ell)$
- $Bid\_Ok(x) \equiv x.bid = Bid(x)$.
- $Agree\_Ok(x) \equiv x.agree = Agree(x)$.
- $Merge\_Dist\_Ok(x) \equiv \forall y : x.merge\_dist[y] = Merge\_Dist(x, y)$
- $Near\_Ok(x) \equiv x.near = Near(x)$.
- $Far\_Ok(x) \equiv x.far = Far(x)$.

- Let $m = x.leader$. Then $Weak\_Cert\_Ok(x)$ is true if the following conditions hold for all ℓ such that $x.weak\_cert[\ell] \neq \perp$.

  1. $\ell \neq \mathrm{m}$
  2. If $x.u[\ell] = x$ then $m < l$.
  3. If $x.v[\ell] = x$ then $m > l$.
  4. If $y \in N(x)$ and $y.leader = m$, then

     (a) $y.u[\ell] = x.u[\ell]$

     (b) $y.v[\ell] = x.v[\ell]$

     (c) $|y.d_u[\ell] - x.d_u[\ell]| \leq 1$

     (d) $|y.d_v[\ell] - x.d_v[\ell]| \leq 1$
  5. If $z \in N(x)$ *and* $y.leader = \ell$, then

     (a) $y.u[m] = x.u[\ell]$

     (b) $y.v[m] = x.v[\ell]$

     (c) $|y.d_u[m] - x.d_u[\ell]| \leq 1$

     (d) $|y.d_v[m] - x.d_v[\ell]| \leq 1$
  6. $$x.d_u = \begin{cases} 0 & \text{if } x.u = x \\ 1 + \begin{cases} \min\{y.d_u[\ell] : y \in N(x) \text{ and } y.leader = m\} \\ \min\{y.d_u[m] : y \in N(x) \text{ and } y.leader = \ell\} \end{cases} & \text{Otherwise} \end{cases}$$
  7. $$x.d_v = \begin{cases} 0 & \text{if } x.v = x \\ 1 + \begin{cases} \min\{y.d_v[\ell] : y \in N(x) \text{ and } y.leader = m\} \\ \min\{y.d_v[m] : y \in N(x) \text{ and } y.leader = \ell\} \end{cases} & \text{Otherwise} \end{cases}$$

Finally, we define the predicates we need for the construction of our algorithm. Each of these is the conjunction of a number of the simpler predicates defined above.

The intuition is that, in order for either Front or Back to run properly, the variables computed by the other two modules must be correct. If not, the algorithm executes Action A2 or A3 of Table 5.6 and starts over.
Once the algorithm starts over in this manner, it will not do so again, but will proceed to completion without error.

$Front\_Ok(x) \equiv$

$Dist\_Ok(x)$ ∧BFS-MIS_$Ok(x) \wedge Beta_Ok(x) \wedge Leader_init_Ok(x) \wedge$

$Leader\_Ok(x) \wedge Weak\_Cert\_Ok(x)$

- $Back\_Ok(x) \equiv$

  $Dist\_Ok(x)$ ∧BFS-MIS_$Ok(x) \wedge Beta_Ok(x) \wedge Leader_init_Ok(x) \wedge$

  $Ldr\_Ok(x) \wedge Border\_Dist\_Ok(x) \wedge Strong\_Cert\_Ok(x) \wedge Bid\_Ok(x) \wedge$

  $Agree\_Ok(x) \wedge Merge\_Dist\_Ok(x) \wedge Near\_Ok(x) \wedge Far\_Ok(x)$

CHAPTER 10

COMPLEXITIES

Lemma 10.1 The time complexity of our algorithm is $O\left(\frac{n^2 diam}{d\_max^2}\right)$

Proof: Preprocess is known to take $O(n)$ rounds [28].

Let w be the current number of weak certificates, the number of pairs of leaders $\{m, \ell\}$ such that there is a weak certificate which certifies that $G(m)$ and $G(\ell)$ are incompatible. Then $w \leq \frac{n}{d\_max}$. Let p be the current number of groups. Define a potential $\Phi = \frac{pn}{d\_max} - w$. Clearly, $0 < \Phi < \frac{pn^2}{d\_max}$. We prove that Φ decreases by at least one during each iteration of the main loop of our algorithm. If no groups are merged during that iteration, w increases, and thus Φ decreases by an integer. Otherwise, the number of groups decreases by at least one, causing the first term of Φ to decrease by at least $\frac{n}{d\_max}$. The second term of Φ can increase by at most $\frac{n}{d\_max}$.

Thus, the number of iterations of the main loop of the algorithm is less than $\frac{pn^2}{d\_max}$. Each iteration takes at most $O(diam)$ rounds, and we are done.

We let H be the maximum cardinality of $H_{d\_max+1}(x)$ for any $x \in X$.

Lemma 10.2 The space complexity of our algorithm is $O(H)$ for each process, where the the space is measured in terms of the number of processes.

Proof: By definition of $H,$ for any process $x,$ $Range(x.dist[\ ])$ has cardinality at most H. $Range(x.grp\_dist[\ ])$ is a subset of $Range(x.dist[\ ]),$ and hence has cardinality at most H.

Every group which borders $G(x)$ contains a process whose distance from x is at most $d\_max + 1$, and thus the number of such groups is less than H. Thus, $Range(x.border\_dist[\ ])$ and $Range(x.strong\_cert[\ ])$ each has cardinality at most H.

The one remaining dynamic array variable of a process x is $x.merge\_dist[\ ]$. The range of that array is at most the cardinality of $G(x) \cup G(\ell),$ where $\ell = x.bid$. Thus, $(Range(x.merge\_dist[\ ])$ has cardinality at most $2H$.

The remaining variables of a process x each take $O(1)$ space. Thus, the space complexity of our algorithm at x is $O(H)$.

Note that $H \leq n$; hence, we can also state that the space complexity of our algorithm is $O(n)$ per process.

CHAPTER 11

COMPETITIVENESS

We define an algorithm for the problem to be C*-competitive* if there is some constant K such that, for any network X, the number of groups in the *d_max*-partition of X computed by the algorithm does not exceed $C \cdot n_{OPT}(X) + K$, where n_{OPT} is the minimum number of groups possible in a *d_max*-partition of X.

A *unit disk graph* is a graph where each node is a point in the plane, and there is an edge between two nodes if and only if the distance between the two points is at most one.

Lemma 11.1 Our algorithm is $2n/d\_max$*-competitive*.

Proof. Every group in the initial partition, other than the one group which contains *Root*_BFS, has at least $d\_max/2$ processes. The number of the groups is thus no greater than $\frac{2(n-1)}{d\_max} + 1$. □

Lemma 11.2 If X is a unit disk graph in the plane, then our algorithm is $4\left(d\_max + \frac{4}{d\_max+2}\right)$-competitive.

Proof. For each $x \in X$, let D_x be the disk of diameter 1 centered at x, which has area $\pi/4$. If $G_1, \dots G_{m_{OPT}}$ is the optimal d_max-partition of X, then each set $U_i = \bigcup_{x \in G_i} D_x$ has diameter at most $d\_max + 1$, and hence, by the isoparametric inequality and Barbier's Theorem, has area at most $\pi(d\_max + 1)^2/4$. It follows that the set $U = U_1 \cup \cdots \cup U_{m_{OPT}} = \bigcup_i D_i$ has area at most $\pi(d\_max + 1)^2 m_{OPT}$.

Let m be the number of groups in the partition computed by our algorithm. Recall MIS, the set of processes of the maximal independent set generated by our algorithm. Let k_{MIS} be the cardinality of MIS. Since $D_x \cap D_y = \emptyset$ for any two distinct $x, y \in$ MIS, we can conclude that the area of U is at least πk_{MIS}. Finally, we recall that every group generated by our algorithm, with the possible exception of the one group containing *Root*_BFS, has at least $(d\_max + 2)/4$ members of MIS. Thus

$$\frac{m(d\_max+2)}{4} + 1 \leq k_{MIS}$$

and $$\frac{\pi k_{MIS}}{4} \leq \frac{\pi (d\_max+1)^2 m_{OPT}}{4}$$

The statement of the lemma follows.

CHAPTER 12

CONCLUSION

We presented the membership management protocol that solves the problem of partitioning a network into groups of bounded diameter.

Given a network of processes X and a constant D, our *self-stabilizing group membership protocol* computes a partition of X, *i.e.,* a set of disjoint connected subgraphs, which we call *groups*, each of diameter no greater than D. In this thesis, a silent self-stabilizing asynchronous distributed algorithm is given for the minimal group partition problem in a network with unique IDs, using the composite model of computation. The algorithm is correct under the *unfair daemon*.

In the unit disk graph X in plane, our algorithm presented in this thesis is $O(d\_max)$-competitive, where d_max is the upper bound on the diameter of any group. That is, the number of groups in the partition constructed by the algorithm is $O(d\_max)$ times the number of groups in the minimum *D-partition*. The time complexity of our algorithm is $O\left(\frac{n^2 diam}{d\_max^2}\right)$, where n is the number of *processes* in the network and *diam* is the diameter of the network. The space complexity of our algorithm is $O(H)$ for each process, where H is the maximum cardinality of (d_max+1)-neighborhood of any process.

Our method is to first construct a breadth-first search (BFS) tree for X, then find a maximal independent set (MIS) of X. Using the MIS and the BFS tree, an initial *D-partition* is constructed, after which groups are merged with adjacent groups until no more mergers are possible. The resulting *D-partition* is minimal.

Mobile ad hoc networks are subject to dynamism where nodes constantly join and leave. The algorithm presented in this thesis can be enhanced in the future to handle the dynamism of network MANETs.

BIBLIOGRAPHY

1. Afek Y, Dolev S. Local stabilizer* 1. Journal of Parallel and Distributed Computing 2002;62(5):745-65.

2. Max-min d-cluster formation in wireless ad hoc networks. IEEE INFOCOM 2000. nineteenth annual joint conference of the IEEE computer and communications societies. proceedings; 2000.

3. Self-organizing systems case study: Peer-to-peer systems. DISC confCiteseer; 2003.

4. Arora A, Gouda M. Distributed reset. IEEE Trans Computer 1994:1026-38.

5. Arora A, Gouda M. Closure and convergence: A foundation of fault-tolerant computing. IEEE Trans Software Eng 1993;19(11):1027.

6. Arora AK. A foundation of fault-tolerant computing. 1992.

7. Self-stabilization by local checking and correction. 32nd annual symposium on foundations of computer science, 1991.

8. Awerbuch B, Patt-Shamir B, Varghese G, Dolev S. Self-stabilization by local checking and global reset. Distributed Algorithms 1994:326-39.

9. Beauquier J, Delaët S, Dolev S, Tixeuil S. Transient fault detectors. Distributed Computing 2007;20(1):39-51.

10. Bottazzi D, Montanari R, Rossi G. A self-organizing group management middleware for mobile ad-hoc networks. Computer Communications 2008;31(13):3040-8.

11. Enabling snap-stabilization. 23rd international conference on distributed computing systems, 2003. proceedings; 2003.

12. Self-stabilizing leader election in optimal space. 10th international symposium on stabilization, safety, and security of distributed systems; 2008.

13. Dijkstra EW. Self-stabilizing systems in spite of distributed control. Commun ACM 1974;17(11):644.

14. Dolev S. Self-stabilization. The MIT press; 2000.

15. Dolev S, Gouda MG, Schneider M. Memory requirements for silent stabilization. Acta Informatica 1999;36(6):447-62.

16. Superstabilizing protocols for dynamic distributed systems. Proceedings of the fourteenth annual ACM symposium on principles of distributed computingACM; 1995.

17. Ducourthial B, Khalfallah S, Petit F. Best-effort group service in dynamic networks. ; 2008.

18. Fault-containing self-stabilizing algorithms. Proceedings of the fifteenth annual ACM symposium on principles of distributed computing ACM; 1996.

19. Heidemann J, Govindan R. An overview of embedded sensor networks ISI TR-2004-594. 2008.

20. Kessels JLW. An exercise in proving self-stabilization with a variant function. Information Processing Letters 1988;29(1):39-42.

21. Liu J, Sacchetti D, Sailhan F, Issarny V. Group management for mobile ad hoc networks: Design, implementation and experiment. ACM; 2005. 199 p.

22. Group management for mobile ad hoc networks: Design, implementation and experiment. Proceedings of the 6th international conference on mobile data management ACM; 2005.

23. Osman H, Taylor H. Managing group membership in ad hoc M-commerce trading systems.

24. Peleg D. Distributed computing: A locality-sensitive approach. Society for Industrial Mathematics; 2000.

25. Strunk JD, Ganger GR. A human organization analogy for self-* systems. Algorithms and Architectures for Self-Managing Systems 2003.

26. Dimple: Dynamic membership protocol for epidemic protocols. IEEE broadnets Citeseer; 2007.

27. Self-stabilization by counter flushing. Proceedings of the thirteenth annual ACM symposium on principles of distributed computing ACM; 1994.

28. Vemula P. Self-stabilizing k-clustering in mobile ad hoc networks. UNLV; 2008.

VITA

Graduate College
University of Nevada, Las Vegas

Mahesh Subedi

Degrees:
    Bachelor of Engineering in Computer Engineering, 2005
    Institute of Engineering, Tribhuvan University, Nepal

Thesis Title: Self-Stabilizing Group Membership Protocol

Thesis Examination Committee:
    Chair Person, Dr. Ajoy K. Datta, Ph.D.
    Committee Member, Dr. John Minor, Ph.D.
    Committee Member, Dr. Lawrence L. Larmore, Ph.D
    Graduate College Representative, Dr. Emma E. Regentova, Ph.D.