

CONVERGENCE SANS SYNCHRONIZATION

By

Arya Tanmay Gupta

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Computer Science – Doctor of Philosophy

2024

ABSTRACT

We currently see a steady rise in the usage and size of multiprocessor systems, and so the community is evermore interested in developing fast parallel processing algorithms. However, most algorithms require a synchronization mechanism, which is costly in terms of computational resources and time.

If an algorithm can be executed in asynchrony, then it can use all the available computation power, and the nodes can execute without being scheduled or locked. However, to show that an algorithm guarantees convergence in asynchrony, we need to generate the entire global state transition graph and check for the absence of cycles. This takes time exponential in the size of the global state space.

In this dissertation, we present a theory that explains the necessary and sufficient properties of a multiprocessor algorithm that guarantees convergence even without synchronization. We develop algorithms for various problems that do not require synchronization. Additionally, we show for several existing algorithms that they can be executed without any synchronization mechanism.

A significant theoretical benefit of our work is in proving that an algorithm can converge even in asynchrony. Our theory implies that we can make such conclusions about an algorithm, by only showing that the local state transition graph of a computing node forms a partial order, rather than generating the entire global state space and determining the absence of cycles in it. Thus, the complexity of rendering such proofs, formal or social, is phenomenally reduced.

Experiments show a significant reduction in time taken to converge, when we compare the execution time of algorithms in the literature versus the algorithms that we design. We get similar results when we run an algorithm, that guarantees convergence in asynchrony, under a scheduler versus in asynchrony. These results include some important practical benefits of our work.

Copyright by
ARYA TANMAY GUPTA
2024

To
my grandfather, Giriraj Nandan,
for his heritage that I am proud to share,
and the rest of my family,
grandma Madhuri Devi,
parents Gopal and Sonali,
brother and comrade Veer Akku Gaurang,
for their love.

ACKNOWLEDGEMENTS

The experiments present in this dissertation were supported through computational resources and services provided by the Institute for Cyber-Enabled Research, Michigan State University.

This dissertation would not have been concluded without the sincere efforts of my advisor, Professor Sandeep S Kulkarni. He motivates me to be thirsty for knowledge and mends me as I go along; I am always enlightened by our discussions. I always have a chance to play with Mathematics and to experiment, in my mind, and on a computer. I can enjoy my work only because of the freedom that he has given to me, and the support that he has shown in all these years. His advice is enlightening, not only with respect to my research, but also to circumstances in my personal life and my academic career. His supervision is outstanding; it is my sole ambition in my academic career to provide a ‘Kulkarni mentorship’ to my students. This is the only way, methinks, through which I will be able to pay forward what I got from him. In addition to Prof Kulkarni, I owe thanks to several other folks. I note some of them in the following paragraphs.

The other members of my doctoral program committee, Drs Abdol-Hossein Esfahanian, Eric Torng and Shaunak D Bopardikar, are some of the most supportive faculty who I worked with during my doctoral program. Their questions and comments about my work have led this dissertation to include some nontrivial lemmas, insights and details. I am thankful to Dr Esfahanian, whose kind supervision and trust have boosted me outstandingly; I am thankful for his hand on my back in moments of immense self-doubt. The support and insightful suggestions that I got from Dr Bopardikar and Dr Torng are unparalleled; they helped me see some interesting nontrivial properties of systems that I present in this dissertation.

Archimedes of Syracuse, Srinivasa Ramanujan, John Forbes Nash Jr, and other extraordinary scientists have taught us that innovation and discovery can strike in the most unusual of circumstances, and so can great enlightening discussions – while teaching a class, at a conference or breakfast, while going to sleep, or even when one is swimming in a pool but cannot help ponder over some mathematical model or problem. I am thankful to Vijay Garg, Maurice Herlihy, Shlomi Dolev, Borzoo Bonakdarpour, Chase Bruggeman, Aljoscha Meyer, for the most valuable discussions and deliberations. Their valuable suggestions and queries have led to shaping this dissertation to a great extent.

I wish to thank my groupmates: Duong Ngoc Nguyen, Gabe Appleton, Jesus Garcia, Raaghav Ravisankar, William Schultze, Ishaan Lagvankar, Luke Sperling, Joel DeBoer, Eliezer Amponsah, for the rigorous discussions that we had in our group meetings. Their fellowship has not only helped me in my work but also kept my spirits high.

A dissertation is not only an extract of concluding intellectual thoughts and efforts, it is also a consequence

of besting non-intellectual battles. I wish to mark some acknowledgements in this realm, next.

The emotion of sadness gives us strength through sustaining (self-)compassion in us (paraphrased: Susan Cain (2022). *Bittersweet*. [1]). I owe thanks to all who mourned with me at my losses and stood by me when I exploited an opportunity or made miscalculations. I am not very social, but even so, I was blessed with some great friends and mentors, whose fellowship provided me with strength and spirit in the most despairing of circumstances.

Before joining MSU, the Graduate School trained us (the incoming TAs) over several scenarios related to pedagogy, research, and life at MSU. This was during the time when the first wave of COVID-19 shook the world. I am grateful for the support that the Graduate School has shown to me, especially Dr Stefanie Baier. My first non-work, non-business, informal, in person interaction post COVID-19 with people happened at OISS Coffee Hour which was resumed in the Fall of 2021 after COVID. I am thankful to the support that Office of International Students and Scholars (OISS) has shown to me, especially Dr Krista Beatty.

We are not so different chemically, biologically, however, historically, we are unique; our thoughts, our actions, our differences, have reasons hidden deep in our stories, our biographies (paraphrased: Oliver Sacks (1985). *The Man who Mistook his Wife for a Hat*. [2]). Friends, colleagues and mentors such as Stefanie Baier, Hima Rawal, Chase Bruggeman, Frost Peanut, Samara Chamoun, Gloria Ashaolu, Tianyi(Titi) Kou-Herrema, Sunia Tanweer, Subhaprad Ash, Abby Hack, Ellen Searle, Nkolay Ivanov, Saviour Kitcher, Sevan Chanakian, DR Mossman, Bryce Carlton, Christian Yanez, Natasha George, have (basically) kept me emotionally and spiritually alive. I cannot thank them enough, and I am not sure whether I will be able to repay them or pay forward the love and completeness that I experienced in their company. There are some friendships where I did not get to interact much during my time at MSU, but they are as fierce as any; I am thankful to Shahrul Akhtar, Anubhav Gupta, Shikhar Goel, Harshita Trikha, Purushottam Verma, for being unconditional.

I am immensely thankful to my family, to whom I dedicate this dissertation, for the care and love that they have shown to me. I am immensely indebted to the patience that my family had to render, when I was not available for them, but they were available for me; I can only hope to repay them. I am thankful to the almighty for keeping my family healthy and safe.

One continuously keeps learning and, even unknowingly, keeps diffusing what he has learned throughout life. I wish to thank my students who I got the opportunity to teach and mentor during my service for 12 semesters, under the title of Graduate Teaching Assistant, with various capacities, at MSU. I have come in contact with some brilliant and heartwarming souls who not only carved the way that I teach but also the way that I think.

Finally, in current times, it is difficult to convince people about the beauty of theoretical work. The

attractive shine of some areas that are rigorously applicable to be a technology has overshadowed the fervor for conducting research just for the sake of the development of science. This includes funding agencies, fellow colleagues, and even a layman. This seems true for theoretical work in any field. Sometimes, we even start to doubt ourselves as to whether our work will be appreciated in the community, just because it is ‘too theoretical’. I acknowledge the austerity, perseverance, integrity, judgement, that I manifested which led me to conclude this dissertation in its current form. I am thankful that I sustained myself in all kinds of circumstances. I am thankful to all the persons, especially those who I have mentioned in this Acknowledgement, who consistently inspired me. I am thankful to Michigan State University for assisting and providing me with every kind of resource that I need to conduct my research.

This Acknowledgement was edited recursively over a period of several months. I sincerely apologize if I forgot to mention someone significant.

- Āryā Tanmay Gupta

TABLE OF CONTENTS

| | |
|--|-----|
| LIST OF SYMBOLS | ix |
| CHAPTER 1 INTRODUCTION | 1 |
| CHAPTER 2 PRELIMINARIES | 8 |
| CHAPTER 3 PARALLELIZING MULTIPLICATION AND MODULO | 17 |
| CHAPTER 4 EVENTUALLY LATTICE-LINEAR ALGORITHMS | 40 |
| CHAPTER 5 FULLY LATTICE-LINEAR ALGORITHMS | 60 |
| CHAPTER 6 PARTIAL ORDER-INDUCING SYSTEMS | 96 |
| CHAPTER 7 RELATED WORK | 116 |
| CHAPTER 8 CONCLUSION | 128 |
| BIBLIOGRAPHY | 133 |
| APPENDIX A OVERVIEW OF GRAPH THEORY AND DISTRIBUTED ALGORITHMS | 140 |
| APPENDIX B PUBLICATIONS FROM THIS DISSERTATION | 151 |

LIST OF SYMBOLS

| | |
|---------------------|---|
| : | such that; where |
| \cup | union |
| \exists | there exists at least one; there exist some |
| $\exists!$ | there exists a unique |
| \forall | for all; for each; for every; for any |
| $\lceil x \rceil$ | ceiling(x); smallest integer greater than or equal to a real number x |
| $\lfloor x \rfloor$ | floor(x); largest integer smaller than or equal to a real number x |
| Adj_i^x | set of nodes within distance x from i , excluding i |
| Adj_i | set of nodes within distance 1 from i , excluding i |
| $dist(x, y)$ | length of shortest path between nodes x and y |
| $E(G)$ | edge set of graph G |
| N_i | $Adj_i \cup \{i\}$ |
| N_i^x | $Adj_i^x \cup \{i\}$ |
| $V(G)$ | vertex set of graph G |
| $Z \setminus Z'$ | (setminus) a set containing elements of Z , excluding elements of Z' |
| $ Z $ | size of the set Z ; number of elements in the set Z |

CHAPTER 1

INTRODUCTION

Given the available technology, increasing the number of computing processors is significantly cheaper than making a single chip more powerful. The development and demand of computing systems are, presently, profoundly affected by this fact, and we see a continuous rise in the usage and size of multiprocessor systems. A multiprocessor system contains multiple computing nodes performing executions to solve one problem. Each node can run one or more processes.

In this dissertation, we assume that each computing node runs only one process to allow maximum parallelization. For this reason, throughout this dissertation, we use the term node for processes and processors alike. A node can be viewed as an independent computing machine, and a multiprocessor system consists of multiple nodes.

In a multiprocessor system, each node may store some data. When the system executes, all nodes read the data that they need through a shared memory or by passing messages.

The goal for the computing nodes is to collectively reach a state where the problem is deemed solved. To achieve this, these nodes need to read the data in a controlled manner so that they synchronize with each other. To illustrate this, consider the problem of graph colouring. An algorithm for this problem can be developed as follows: if node i reads that it has a conflicting colour with at least one of its neighbours, then it changes its colour to the minimum possible available colour. If this algorithm is run in an interleaving fashion (where one node executes at a time), then it will converge to a state where all nodes have non-conflicting colours.

In a uniprocessor execution (or a multiprocessor execution where only one node executes at a time), this algorithm runs correctly because the global states form a directed acyclic graph (DAG), and all the sink nodes of this DAG are optimal states. This property is also necessary and sufficient for the correctness of an algorithm that must reach an optimal state and then terminate/stutter. However, if the above algorithm is run on a multiprocessor system in asynchrony, it may never converge. This can be explained by the following example. Consider a graph with only two nodes, i and j , connected to each other by an edge. Consider that both i and j are initialized with colour value 1. Suppose that i reads the value of j and decides to change its colour to 2. However, if j reads the state of i asynchronously before the changes made by i are reflected, it will also decide to change its colour to 2. Similarly, in the next step, both nodes may decide to change their colour to 1. This can continue forever and the system may never converge.

Convergence cannot be guaranteed, from an arbitrary state, in the above system because of the presence of a cycle in the global state transition graph, which, in turn, is induced due to the race conditions arising

among neighbouring nodes. Deployment of proper synchronization mechanisms (in this case, local mutual exclusion) eliminates such behaviour.

From above, we observe: (1) under the assumption of synchronization, analysis of system behaviour and design of algorithms becomes easier, and (2) if such assumptions are removed, then we may observe cyclic behaviour in state transitions, making it tedious to analyze the system, and in some cases, even preventing the system from convergence. Thus, designing an algorithm that guarantees convergence without synchronization is not trivial.

There are various synchronization primitives considered in the literature. A common synchronization primitive is to use a *scheduler/daemon* that identifies how/when nodes can execute. The synchronization model used in the above-discussed algorithm for graph colouring uses the *central scheduler*, which chooses only one node per time step to execute. Other than this, there are other scheduler-based synchronization models, e.g., a *distributed scheduler* chooses one or more nodes, possibly arbitrarily, per time step, and a *synchronous scheduler* chooses all the nodes in each time step. All schedulers implicitly assume a barrier requirement that one step has to complete before the next step can begin. This means that if multiple nodes are executing a step then it is necessary to wait until all executing nodes finish their step before starting the next one.

Apart from enforcing the above-discussed scheduler-based synchronization primitives, there are other synchronization primitives like local mutual exclusion, token ring and semaphores. In all such models, the idea is to prohibit certain processes to execute, depending on the availability of resources or semantics of the subject algorithm. This assumption restricts the usage of resources and flow of data, and thus, makes the design of algorithms easy, ensuring correctness.

Enforcing synchronization introduces an overhead, which can be very costly in terms of computational resources and time. For this reason, the community is interested in developing algorithms that require minimum possible synchronization. If an algorithm can be executed in asynchrony, then it can use all the available computation power, and the nodes can execute without being scheduled or locked. If we understand the underlying behaviour of such algorithms, then we can easily analyze if an algorithm possesses this quality, and if not, we can make minimal changes to it and transform it into an algorithm that does not require synchronization.

While there have been instances of algorithms that are correct under asynchrony (e.g., [3–5]), we do not have a model that captures and explains the behaviour of an arbitrary algorithm that converges in asynchrony. Hence, in this dissertation, we focus on developing a theory for multiprocessor algorithms that guarantees convergence under asynchrony. Specifically, we validate the following thesis statement:

Synchronization is not required in a multiprocessor system which can identify, in every suboptimal global state, at least one node whose local state will indefinitely prevent convergence.

Organization of the Chapter

We validate the above thesis statement by building on a recent work by Garg [6]. We briefly discuss Garg’s model of lattice-linearity in Section 1.1. In Section 1.2, we enumerate the contributions of this dissertation. In Section 1.3, we discuss the applications of the problems that we study in this dissertation. Section 1.4 outlines the organization of the following chapters in this dissertation.

1.1 Lattice-Linearity

The recent introduction of lattice-linearity (by Garg, SPAA 2020) [6] has shown that algorithms can be developed in such a way that the nodes can execute asynchronously.

A critical observation from [6] is that if the global states form a lattice, then, under some additional constraints, an algorithm traversing that lattice is fully tolerant to asynchrony. The key idea of lattice-linearity is that in such systems, a node can determine that its local state is not feasible in any reachable optimal global state, so, it has to change its state to reach an optimal state. Thus, if node i changes its state from $i[st]$ to $i[st']$ it never revisits state $i[st]$ again.

Since it is guaranteed for a node in a violating state that its current state has to be rejected permanently, and that no optimal global state can be reached in its current local state, it can change its state even if it is relying on the old values of its neighbours. This allows the nodes to run without synchronization and the system is yet guaranteed to reach an optimal state. There are some additional constraints regarding what the next chosen local state $i[st']$ will be, once i rejects its current state $i[st]$. These constraints are problem-dependent; we discuss these constraints, in detail, in the following chapters, according to their relevance to the subject problem and the algorithm being studied.

As a consequence of the property that a local state once rejected is never visited again, the local states visited by each individual node form a total order. The lattice structure induced among the global states is a consequence of this total order induction in the local state transition graph.

In [6], Garg studies a restricted class of problems, which we call *lattice-linear problems*. In such problems, the nodes that are in a violating local state can be distinctly determined. In addition, the problems studied in [6] do not allow *self-stabilization*.

1.2 Contributions of the Dissertation

We first study whether there exist any lattice-linear problems that allow self-stabilization. To this end, we show that the parallel processing version developed for Karatsuba’s multiplication algorithm (cf. [7]) by Cesari and Maeder [8] guarantees convergence without synchronization. This algorithm is lattice-linear and

is self-stabilizing. We study multiple algorithms for multiplication and modulo operations that are lattice-linear, and effectively, they guarantee convergence in asynchrony. This study is presented in Chapter 3; a preliminary version of this chapter appeared in SSS 2023.

The problem that we study next is whether we can develop algorithms for non-lattice-linear problems (problems in which violating nodes cannot be determined in a suboptimal global state) that converge in asynchrony. To this end, we observe that, under problem-specific constraints, convergence can be guaranteed in non-lattice-linear problems algorithmically even if one or more lattices are induced only in a subset of the state space. This leads to the introduction of eventually lattice-linear algorithms, and we develop eventually lattice-linear algorithms for service-demand based minimal dominating set, minimal vertex cover, maximal independent set, graph colouring and 2-dominating set problems. This study is presented in Chapter 4; a preliminary version of this chapter appeared in SSS 2021 and the full chapter was published in JPDC 2024. We also present some experimental results, comparing the runtime of our algorithm for maximal independent set with other algorithms in the literature. Experimental results show that our algorithm converges much faster as compared to other algorithms.

In eventually lattice-linear algorithms, lattices are induced only among a subset of the global states, so a developer must guarantee that the system, initialized in an arbitrary or specified state, is (1) guaranteed to reach a state in one of the lattices, and then (2) guaranteed to reach an optimal state. Thus, the problem that we study next is whether it is possible to induce lattices in the entire state space if the underlying problem is a non-lattice-linear problem. To this end, we observe that lattices can be induced algorithmically in the entire state space in non-lattice-linear problems. Thus, we introduce fully lattice-linear algorithms, and develop algorithms for minimal dominating set, graph colouring, minimal vertex cover and maximal independent set problems. We also present a parallel-processing lattice-linear 2-approximation algorithm for the vertex cover problem. This study is presented in Chapter 5; a preliminary version of this chapter appeared in SRDS 2023. We transform the lattice-linear algorithm for minimal dominating set (which is originally a distance-4 algorithm) to a distance-1 algorithm. We present some experimental results, comparing the runtime of our algorithms for minimal dominating set with other algorithms in the literature. Experimental results show that the distance-1 algorithm that we develop for minimal dominating set converges much faster as compared to other algorithms.

We also show for an algorithm developed by Goswami et. al (SSS, 2022) that it is lattice-linear, and is thus, tolerant to asynchrony. This algorithm was developed to solve the gathering problem on a finite number of robots on an infinite triangular grid. The authors of this algorithm originally assumed a distributed scheduler. We also show that this algorithm converges in $2n$ rounds, which is less than the time complexity originally showed (i.e., $2.5n$ rounds). Apart from this, we find that some guards used in the original algorithm

are redundant, so we remove them and present a new updated algorithm, which uses only a subset of guards from the original algorithm. This study is presented in Chapter 5 (Section 5.8); this study appeared in EDCC 2024.

We have that lattice induction is sufficient to allow asynchrony, but it is not a necessary condition. In other words, all algorithms that induce a lattice structure in the state space (that results from the induction of total order in the local state transition graph) allow asynchrony, but given an arbitrary algorithm that can tolerate asynchrony, it is possible that it is not lattice-linear. Thus, we investigate to find the necessary and sufficient conditions under which an algorithm guarantees convergence in asynchrony. To this end, we find that asynchrony can be allowed if and only if the local states visited by individual nodes form a partial order. Due to the partial order induced among the local states, the global states form a directed acyclic graph (DAG). Effectively, we introduce the classes of DAG-inducing problems and DAG-inducing algorithms. We study the dominant clique problem and the shortest path problems as DAG-inducing problems, and the maximal matching problem as a non-DAG-inducing problem. We provide algorithms for all these problems. This study is presented in Chapter 6.

It is noteworthy from above that problems such as dominant clique and shortest path cannot be modelled within the class of lattice-linear problems, and maximal matching cannot be modelled within the class of non-lattice-linear problems. The algorithms that we present for these problems cannot be modelled within the class of lattice-linear algorithms. This is because here, the local state transition graph of each individual node forms a partial order, and this behaviour cannot be modelled within a discrete structure such as the total order.

For several problems we study in this dissertation, we develop multiple algorithms. In such cases, we analyze those algorithms in such a way that the differences in their behaviour become clear.

The algorithms that we develop theoretically converge faster as compared to the other algorithms in the literature – in terms of the number of moves, or the order of time. We have conducted several experiments to investigate the benefit of asynchrony. Our algorithms that were put to this test are observed to run faster than other algorithms in the literature. We also observe that algorithms that guarantee convergence in asynchrony run faster in asynchrony as compared to the same algorithm running under a scheduler.

1.3 Applicability of Our Work

The major focus and benefit of this dissertation is in developing algorithms that guarantee convergence in asynchrony, and in developing a theory that explains the properties of such algorithms.

A significant theoretical benefit of our work is in proving that an algorithm can converge even in asynchrony. Our theory implies that we can show that an algorithm can tolerate asynchrony, by only showing that the local state transition graph of an arbitrary computing node forms a partial order, rather than

generating the entire global state space and determining the absence of cycles in it. Thus, the complexity of rendering such proofs, formal or social, is phenomenally reduced. We also study the time complexity of an arbitrary asynchrony-tolerant algorithm, and how it is tied to the complexity class of the subject problem.

Experiments show a significant reduction in time taken to converge, when we compare the execution time of algorithms in the literature versus the algorithms that we design. We get the same results when we run an algorithm, that guarantees convergence in asynchrony, under a scheduler versus in asynchrony. These results include some remarkable practical benefits of our work.

Some applications of the specific problems studied in this dissertation are listed as follows. Dominating set is applied in communication and wireless networks where it is used to compute the virtual backbone of a network. Vertex cover is applicable in (1) computational biology, where it is used to eliminate repetitive DNA sequences – providing a set covering all desired sequences, and (2) economics, where it is used in camera instalments – it provides a set of locations covering all hallways of a building. Independent set is applied in computational biology, where it is used in discovering stable genetic components for designing engineered genetic systems. Graph colouring is applicable in (1) chemistry, where it is used to design storage of chemicals – a pair of reacting chemicals are not stored together, and (2) communication networks, where it is used in wireless networks to compute radio frequency assignment.

Matching has applications in numerous areas including social networks. Shortest path problem has applications in network routing and geographical route navigation. Dominant clique problem has applications in social networks and ecology (cf. [9]).

The applications of integer multiplication include the computation of power, matrix products which has applications in a plethora of fields including artificial intelligence and game theory, the sum of fractions and coprime base. Modular arithmetic has applications in theoretical mathematics, where it is heavily used in number theory and various topics (e.g., groups, rings, fields, knots) in abstract algebra. Modular arithmetic also has applications in applied mathematics, where it is used in computer algebra and cryptography. It has applications also in chemistry and the visual and musical arts. In many of these applications, the value of the divisor is fixed. In addition to these applications, it is needless to note that multiplication and modulo are among the fundamental mathematical operations.

1.4 Organization of the Dissertation

The following chapters are organized as follows. In Chapter 2, we discuss preliminary symbols and definitions that we utilize in the chapters that follow. In Chapter 3, we study the properties of lattice-linearity in the parallel processing version for Karatsuba’s multiplication algorithm (cf. [7]) present in [8]. We study some more algorithms for multiplication and modulo operations that are lattice-linear. In Chapter 4, we introduce the class of *eventually lattice-linear algorithms*, and present example algorithms for several problems, along

with some experimental results. In Chapter 5, we introduce the class of *fully lattice-linear algorithms*, and present example algorithms for several problems, along with some experimental results. Chapter 6 investigates the conditions that are necessary and sufficient, under which an algorithm can guarantee convergence even in asynchrony. Like the previous chapters, in this chapter also, we present asynchrony-tolerant example algorithms for several problems. We discuss the related work in Chapter 7. We conclude in Chapter 8.

This dissertation is written such that we assume that a reader is aware of some basic results in graph theory and distributed systems. For a reader who is not aware of these topics, we provide a preliminary overview of these subjects in Appendix A. We provide a list of peer-reviewed publications that emerged from this dissertation in Appendix B.

CHAPTER 2

PRELIMINARIES

Most algorithms that we study in this dissertation are graph algorithms where the input is a graph G , $V(G)$ is the set of its vertices, $E(G)$ is the set of its edges. In our algorithms, each computation node will simulate a distinct vertex of a graph, so we use the term nodes for vertices; in the rest of this dissertation, $V(G)$ will be called a set of the nodes of graph G ; this can also be visualized as each vertex of G acting as a computation node.

For the graph G , $n = |V(G)|$, and $m = |E(G)|$. For a node $i \in V(G)$, Adj_i is the set of nodes adjacent to i , and Adj_i^x is the set of nodes within distance- x from i , excluding i . Neighbourhood of i is adjacency of i , including i , i.e., $N_i = Adj_i \cup \{i\}$ and $N_i^x = Adj_i^x \cup \{i\}$. Degree of a node i is the number of nodes connected to i by an edge, i.e., $deg(i) = |Adj_i|$. The length of shortest path from i to node j is denoted by $dis(i, j)$.

For a finite natural number x , $[1 : x]$ denotes a sequence of all natural numbers from 1 to x .

2.1 Modeling Distributed Systems

A parallel/distributed algorithm consists of nodes where each node is associated with a set of variables. *Local state* of a node i is a sequence of values of all its variables. A *global state*, say s , is a sequence of local states of all nodes. Effectively, we represent s as a vector, where $s[i]$ itself is a vector of the variables of node i . We denote the *state space* by S , which is the set of all global states that a given system can obtain.

Each node in $V(G)$ is associated with rules. Each rule at node i checks values of the variables of the nodes in N_i^x (where the value of x depends on the subject problem and acting algorithm) and updates the variables of i . A *rule* at a node i is of the form $g \rightarrow a_c$, where the *guard* g is a Boolean expression over variables in N_i^x and the *action* a_c is a set of instructions that updates the variables of i if g is true. A node is *enabled* iff at least one of its guards is true, otherwise it is *disabled*.

A *move* is an event in which an enabled node updates its variables. A *round* is a sequence of events in which every node evaluates its guards at least once, and makes a move iff it is enabled.

In many algorithms presented in this dissertation, at a time, atmost one guard per node holds true. In the case that more than one guard is true and more than one action is to be executed, the actions will be executed in the order in which they are written, and in the order in which their respective guards are written. If a node executes more than one actions in a single move, then all those actions will take into consideration the updations made by the actions that were executed before them.

The *state transition system* \mathcal{S} on the state space S is a discrete structure that defines all the possible transitions that can take place among the states of S . Under a given algorithm A , \mathcal{S} is a directed graph such that $V(\mathcal{S}) = S$, and $E(\mathcal{S}) = \{(s, s') | \langle s, s' \rangle \text{ is a state transition under } A\}$.

We assume S_o to be the set of *optimal* global states: the system is deemed converged once it reaches a state in S_o . All other global states are *suboptimal*. An algorithm A is *self-stabilizing* with respect to the subset S_o of S iff it satisfies the following properties: (1) *convergence*: starting from an arbitrary state, any sequence of computations of A reaches a state in S_o , and (2) *closure*: any computation of A starting from S_o always stays in S_o . A is a *silent* self-stabilizing algorithm if no node makes a move once a state in S_o is reached.

We use the *work complexity* of an algorithm with respect to time consumed; it is the summation of the time taken by all nodes to execute whenever they were enabled. The term *time complexity* is also used with respect to time consumed, however, this term, on the other hand, describes the order of time taken for an algorithm to reach an optimal state; it does not provide any information to the reader/observer about the work complexity of an algorithm, but it tells how much time a distributed system as a whole will take to converge. As an example, if an algorithm executes actions 1, 2, and 3, each taking 1 time unit, such that actions 1 and 2 are executed concurrently and action 3 is executed only after they finish, then the time complexity is 2 units and the work complexity is 3 units.

2.1.1 Execution Without Synchronization.

Typically, we view the *computation sequence* of an algorithm as a sequence of global states $\langle s_0, s_1, \dots \rangle$, where s_{t+1} ($t \geq 0$) is obtained by executing some action by one or more nodes (as decided by the scheduler) in s_t . For the sake of discussion, assume that only node i executes in state s_t , and it has only one variable. The computation prefix upto s_t is $\langle s_0, s_1, \dots, s_t \rangle$. The state that the system traverses to after s_t is s_{t+1} . Under proper synchronization, i would evaluate its guards on the *current* local states of its distance- x ($x \geq 1$) neighbours in s_t , resulting in the system reaching s_{t+1} .

We assume that an observer oracle is able to take a consistent snapshot of the system instantly without having to stop the executions of the nodes. However, the computing nodes may not know the fresh local states of other nodes. To understand how the execution works in asynchrony, let $s[j]$ be the local state of node j in state s . If i executes in asynchrony, then since the reads performed by the nodes, as well as their movements, are not coordinated with each other, the read operation performed by i may return older values. As a result, i views the global state that it is in to be s' where, for an arbitrary node $j \neq i$, $s'[j] \in \{s_0[j], s_1[j], \dots, s_t[j]\}$. This means that i can read older local states of other nodes from arbitrarily older global states.

In the case that i may read older values, s_{t+1} is evaluated as follows. If all guards of i evaluate to false in global state s' , then the system will continue to remain in state s_t , i.e., $s_{t+1} = s_t$. If some guard g evaluates to true in s' then i will execute its corresponding action a_c . Here, we have the following observations: (1) $s_{t+1}[i]$ is the state that i obtains after executing an action in s' , and (2) $\forall j \neq i, s_{t+1}[j] = s_t[j]$.

2.1.2 Variations of Asynchrony

In this dissertation, we are interested in two models: arbitrary asynchrony (AA) and asynchrony with monotonous read (AMR). In the AA model, as described above, a node can read old values of other nodes arbitrarily; here, we only assume that if some information is sent from a node, it eventually reaches the target node. Similar to AA, in AMR, the nodes execute asynchronously. However, the AMR model adds another restriction: the values of variables of other nodes are read/received in the order in which they were updated/sent.

Algorithms present in this dissertation that require AMR model are also guaranteed to converge when there is a much more relaxed requirement that given a pair of arbitrary nodes i and j , node i can read arbitrarily old values of j (as allowed by AA), but i will eventually stop reading/receiving the values that j obtained but rejected. However, in this more relaxed model, our proofs that describe the time complexity and other characteristics of such algorithms are, clearly, not valid.

In both AA and AMR, node i reads the most recent state of itself.

Our algorithms are independent of, and allow, both the message-passing model and the shared memory model, and our correctness and time-complexity proofs remain correct in both these models. We do not assume any node failures or byzantine behaviours, other than the nodes running without synchronization.

2.1.3 Simple Examples

We discuss two example problems and one algorithm, each, to solve them. Our intent here is to show the difference between the working of algorithms that guarantee convergence in asynchrony, versus the algorithms that cannot make this guarantee. Specifically, we show how the naive algorithm for graph colouring that we discuss in the Introduction does not guarantee convergence in asynchrony. We introduce the max problem, and we discuss an algorithm for it that converges in asynchrony.

Example 2.1. Colouring. *Consider the example algorithm for the graph colouring problem discussed in the Introduction. This algorithm, if run under a central scheduler, guarantees convergence. If the initial state is $\langle 1, 1 \rangle$, then depending on which node executes first, it is guaranteed to converge to an optimal state (see Figure 2.1 (a), solid lines going out from $\langle 1, 1 \rangle$ and from $\langle 2, 2 \rangle$). However, if this algorithm is run in AA or AMR model, then it does not guarantee convergence. In AA model, e.g., all possible global state transitions are possible, depending on the oldness of the values that the nodes are reading from each other (see Figure 2.1 (a), all dotted and solid lines). \square*

Example 2.2. Max. *Now consider the max problem on 3 nodes, and let the initial state be $\langle 1, 2, 3 \rangle$. Consider the following naive algorithm: if a node i reads that there is another node j whose value is greater than that of i , then i changes its own value to be equal to the value of j . If this algorithm is run under a central scheduler,*

then convergence is guaranteed (possible transitions are represented by solid edges in Figure 2.1 (b)). Notice that if this algorithm is run in AMR or AA model, then also convergence is guaranteed. However, AA or AMR model result in additional transitions. Specifically, state $\langle 1, 3, 3 \rangle$ has a predecessor $\langle 1, 2, 3 \rangle$, so in state $\langle 1, 3, 3 \rangle$, node 1 can read an old local state of node 2 (i.e., 2) and not the current state of node 2 (i.e., 3). Thus, another transition $\langle 1, 3, 3 \rangle \rightarrow \langle 2, 3, 3 \rangle$ is allowed in this case (presented as a dashed edge in Figure 2.1 (b)). \square

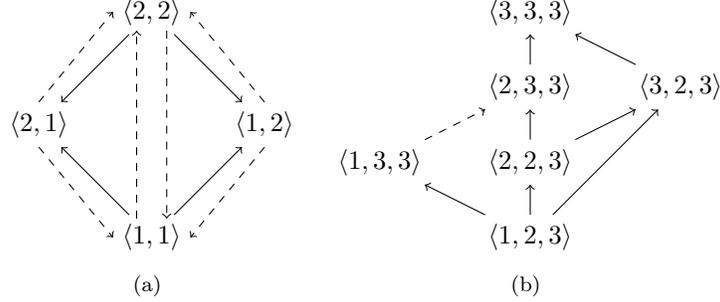


Figure 2.1: (a) State transitions of a naive 2-node algorithm for graph colouring. (b) State transitions under a naive algorithm for the max problem.

2.2 Embedding a \prec -lattice in Global States

In this section, we discuss the structure of a lattice in the state space which, under proper constraints, allows an algorithm to converge to an optimal state. To describe the embedding, we define a total order \prec_l ; all local states of a node i are totally ordered under \prec_l . Using \prec_l , we define a partial order \prec_g among global states as follows.

We say that $s \prec_g s'$ iff $(\forall i : s[i] = s'[i] \vee s[i] \prec_l s'[i]) \wedge (\exists i : s[i] \prec_l s'[i])$. Also, $s = s'$ iff $\forall i : s[i] = s'[i]$. For brevity, we use \prec to denote \prec_l and \prec_g : \prec corresponds to \prec_l while comparing local states, and \prec corresponds to \prec_g while comparing global states. We also use the symbol ' \succ ' which is such that $s \succ s'$ iff $s' \prec s$. Similarly, we use symbols ' \preceq ' and ' \succeq '; e.g., $s \preceq s'$ iff $s = s' \vee s \prec s'$. We call the lattice, formed from such partial order, a \prec -lattice.

Definition 2.1. \prec -lattice. Given a total relation \prec_l that orders the values of $s[i]$ (the local state of node i in state s), for each i , the \prec -lattice corresponding to \prec_l is defined by the following partial order: $s \prec s'$ iff $(\forall i : s[i] \preceq_l s'[i]) \wedge (\exists i : s[i] \prec_l s'[i])$.

A \prec -lattice constraints how global states can transition among one another: a global state s can transition to state s' only if $s \prec s'$.

In the \prec -lattice discussed above, we can define the meet and join of two states in the standard way: the meet (respectively, join), of two states s_1 and s_2 is a state s_3 where $\forall i : s_3[i]$ is equal to $\min(s_1[i], s_2[i])$

(respectively, $\max(s_1[i], s_2[i])$), where for a pair of local states x and y , if $x \prec_l y$, then $\min(x, y) = \min(y, x) = x$ and $\max(x, y) = \max(y, x) = y$. We are interested in the \prec -lattices where a join can be found for any pair of global states, however, a meet may not be found for some but not all the pairs of global states, the examples of which we study, in this dissertation, in the following chapters. This makes a \prec -lattice an incomplete lattice.

By varying \prec_l that identifies a total order among the states of a node, one can obtain different lattices. A \prec -lattice, embedded in the state space, is useful for permitting the algorithm to execute asynchronously. Under proper constraints on how the lattice is formed, convergence is ensured. We discuss these constraints in the next section.

2.3 Introduction to Lattice-Linearity

Lattice-linearity has been shown to allow asynchrony among the computing nodes whenever it is induced in problems. The key idea of lattice linearity is that if a node can determine if its current state is not feasible in any optimal global state, then it can perform executions even based on old and inconsistent values, i.e., without needing any synchronization. We call such nodes *impedensable*¹ nodes (an *impediment* to progress if does not execute, *indispensable* to execute for progress).

Definition 2.2. [6] *Impedensable node.* $\text{IMPEDENSABLE}(i, s, \mathcal{P}) \equiv \neg\mathcal{P}(s) \wedge (\forall s' \succ s : s'[i] = s[i] \Rightarrow \neg\mathcal{P}(s'))$.

Example 2.Max: Continuation 1. *Under the algorithm for the max problem present in Section 2.1.3, we notice that in any suboptimal global state s , an impedensable node i is a node that does not have the highest value stored in it. For example in state $\langle 2, 2, 3 \rangle$ node 1 and node 2 are impedensable. If any of these nodes retains its state, then it will prevent the system from convergence.* \square

Example 2.Colouring: Continuation 1. *Notice that in the graph colouring problem, an impedensable node cannot be determined. This is because, in any given global state and any chosen node, convergence can be achieved without changing the local state of that node.* \square

The problems in which nodes can make such decisions are called *lattice linear problems*. In such problems, a node once discards its local state, by definition, discards it forever. This can form a total order among the states of a node, resulting in the induction of a lattice among the global states. In such a system, because a local state that is discarded is never revisited, computing nodes can run without synchronization and the system is guaranteed to converge correctly even when the nodes read old values of each other.

¹The term ‘impedensable’ is similar to the notion of a node being *forbidden* introduced in [6]. This word itself comes from predicate detection background [10]. We changed the notation to avoid the misinterpretation of the English meaning of the word ‘forbidden’.

Now, because convergence is guaranteed in such systems and nodes do not revisit the local states that they reject, in any suboptimal global state, there must be at least one node that must change its state, i.e., there must be at least one impedensable node in every suboptimal global state.

Definition 2.3. Impedensable global state. $IMPEDENSABLE(s, \mathcal{P}) \equiv \exists i : IMPEDENSABLE(i, s, \mathcal{P})$.

Example 2.Max: Continuation 2. Under the algorithm for the max problem present in Section 2.1.3, we notice that in any suboptimal global state, there is at least one impedensable node. All suboptimal global states are impedensable global states. \square

Example 2.Max: Continuation 3. Under the algorithm for the max problem presented in Section 2.1.3, the local states form a total order. E.g., for node 1, this order is $1 \rightarrow 2 \rightarrow 3$. Since node 2 is initialized in local state 2, the total order induced among its local states is $2 \rightarrow 3$.

Since all nodes follow the same algorithm, the partial order formed among the local states is essentially the same; its starting point only depends on the local state of initialization for each individual node.

Due to the total order formed among the local states of each individual node, the global states form a \prec -lattice, which is shown in Figure 2.1 (b). \square

In this section, we discuss *lattice-linear problems*, i.e., the problems where the description of the problem statement creates the lattice structure. Such problems can be represented by a predicate under which the states in S form a lattice. Such problems have been discussed in [6, 11, 12].

A *lattice-linear problem* P can be represented by a predicate \mathcal{P} such that if any node i is violating \mathcal{P} in a state s , then it must change its state. Otherwise, the system will not satisfy \mathcal{P} . Let $\mathcal{P}(s)$ be true iff the global state s satisfies \mathcal{P} . A node violating \mathcal{P} in s is called an *impedensable* node.

If a node i is impedensable in state s , then in any state s' such that $s' \succ s$, if the state of i remains the same, then the algorithm will not converge. Thus, predicate \mathcal{P} induces a total order among the local states visited by a node, for all nodes. Consequently, the discrete structure that gets induced among the global states is a \prec -lattice, as described in Definition 2.1. We say that \mathcal{P} , satisfying Definition 2.2, is *lattice-linear* with respect to that \prec -lattice. \mathcal{P} is used by the nodes to determine if they are impedensable, using Definition 2.2.

Definition 2.4. [6]Lattice-linear predicate. \mathcal{P} is an LLP with respect to a \prec -lattice induced among the global states iff $\forall s \in S : \neg \mathcal{P}(s) \Rightarrow \exists i : IMPEDENSABLE(i, s, \mathcal{P})$.

Example 2.Max: Continuation 4. The predicate, governing the algorithm for the max problem presented in Section 2.1.3, can be noted as follows.

$$IMPEDENSABLE-MAX(i) \equiv \exists j : j[val] > i[val].$$

When this predicate is combined with the actions imposed the algorithm, the state transition graph, as presented in Figure 2.1 (b), is formed. This transition graph is a \prec -lattice. \square

Now we complete the definition of lattice-linear problems. In a lattice-linear problem P , given any suboptimal global state s , P specifies all and the only nodes which cannot retain their local states. \mathcal{P} is thus designed conserving this nature of the subject problem P , following Definitions 2.2 and 2.4.

Definition 2.5. Lattice-linear problems. Problem P is lattice-linear iff there exists a predicate \mathcal{P} and a \prec -lattice such that

- P is deemed solved iff the system reaches a state where \mathcal{P} is true,
- \mathcal{P} is lattice-linear with respect to the \prec -lattice induced among the states in S , i.e., $\forall s : \neg\mathcal{P}(s) \Rightarrow \exists i : \text{IMPEDENSABLE}(i, s, \mathcal{P})$, and
- $\forall s : (\forall i : \text{IMPEDENSABLE}(i, s, \mathcal{P}) \Rightarrow (\forall s' : \mathcal{P}(s') \Rightarrow s'[i] \neq s[i]))$.

Remark: A \prec -lattice, induced under \mathcal{P} , allows asynchrony because if a node, reading old values, reads the current state s as s' , then $s' \prec s$. So $\neg\mathcal{P}(s') \Rightarrow \neg\mathcal{P}(s)$ because $\text{IMPEDENSABLE}(i, s', \mathcal{P})$ and $s'[i] = s[i]$.

Example 2.Max: Continuation 5. The max problem, as discussed in Section 2.1.3 is a lattice-linear problem. This is because, in any suboptimal state, we can determine all the nodes that are impedensable. \square

Example 2.Colouring: Continuation 2. Since an impedensable node cannot be determined in the graph colouring problem, we put the graph colouring problem in the class of non-lattice-linear problems. We discuss more on such problems later in this dissertation (in Section 3.6, Chapter 4 and Chapter 5). \square

Definition 2.6. Self-stabilizing lattice-linear predicate. Continuing from Definition 2.5, \mathcal{P} is a self-stabilizing lattice-linear predicate if and only if the supremum of the lattice, that \mathcal{P} induces, is an optimal state.

Note that a self-stabilizing lattice-linear predicate \mathcal{P} can also be true in states other than the supremum of the \prec -lattice.

Example 2.3. SMP. We describe a lattice-linear problem, the stable (man-optimal) marriage problem (SMP) from [6]. In SMP, all men (respectively, women) rank women (respectively men) in terms of their preference (lower rank is preferred more). A man proposes to one woman at a time based on his preference list, and the proposal may be accepted or rejected.

A global state is represented as a vector s where the vector $s[i]$ contains a scalar that represents the rank of the woman, according to the preference of man i , whom i proposes.

SMP can be defined by the predicate

$$\mathcal{P}_{SMP} \equiv \forall m, m' : m \neq m' \Rightarrow s[m] \neq s[m'].$$

\mathcal{P}_{SMP} is true iff no two men are proposing to the same woman. A man m is impedensable iff there exists m' such that m and m' are proposing to the same woman w and w prefers m' over m . Thus,

$$\text{IMPEDENSABLE-SMP}(m, s, \mathcal{P}_{SMP}) \equiv \exists m' : s[m] = s[m'] \wedge \text{rank}(s[m], m') < \text{rank}(s[m], m).$$

If m is impedensable, he increments $s[m]$ by 1 until all his choices are exhausted. Following this algorithm, an optimal state, i.e., a state where the sum of regret of men is minimized, is reached. \square

A key observation from the stable marriage problem (SMP) and other problems from [6] is that the states in S form *one* lattice, which contains a global infimum ℓ and possibly a global supremum u i.e., ℓ and u are the states such that $\forall s \in S, \ell \preceq s$ and $u \succeq s$.

Example 2.SMP: continuation 1. As an illustration of SMP, consider the case where we have 3 men and 3 women. The lattice induced in this case is shown in Figure 2.2. In this figure, every vector represents the global state s such that $s[i]$ represents the rank of the woman, according to the preference of man i , whom i proposes. The algorithm begins in the state $\langle 1, 1, 1 \rangle$ (i.e., each man starts with his first choice) and continues its execution in this lattice. The algorithm terminates in the lowest state in the lattice where no node is impedensable. \square

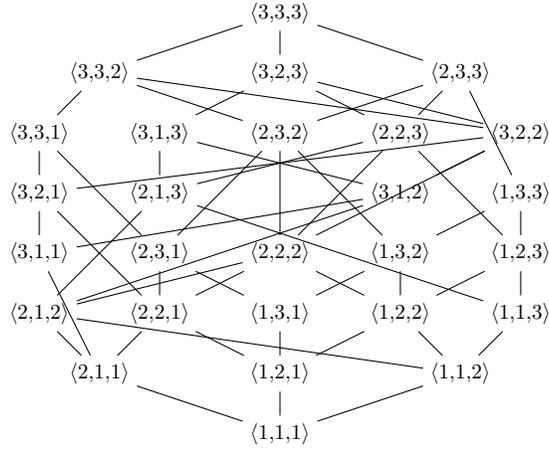


Figure 2.2: Lattice for SMP with 3 men and 3 women; $\ell = \langle 1, 1, 1 \rangle$ and $u = \langle 3, 3, 3 \rangle$. Transitive edges are not shown for brevity.

In SMP and other problems in [6], the algorithm needs to be initialized to ℓ to reach an optimal solution. If we start from a state s such that $s \neq \ell$, then the algorithm can only traverse the lattice from s . Hence, upon termination, it is possible that the optimal solution is not reached. In other words, such algorithms cannot be self-stabilizing unless u is optimal.

Example 2.SMP: continuation 2. Consider that men and women are $M = \langle A, J, T \rangle$ and $W = \langle K, Z, M \rangle$ indexed in that sequence respectively. Let that proposal preferences of men are $A = \langle Z, K, M \rangle$, $J = \langle Z, K, M \rangle$ and $T = \langle K, M, Z \rangle$, and women have ranked men as $Z = \langle A, J, T \rangle$, $K = \langle J, T, A \rangle$ and $M = \langle T, J, A \rangle$. The optimal state (starting from $\langle 1, 1, 1 \rangle$) is $\langle 1, 2, 2 \rangle$. Starting from $\langle 1, 2, 3 \rangle$, the algorithm terminates at $\langle 1, 2, 3 \rangle$ which is not optimal. Starting from $\langle 3, 1, 2 \rangle$, the algorithm terminates declaring that no solution is available. □

CHAPTER 3

PARALLELIZING MULTIPLICATION AND MODULO

We observe that the community is evermore interested in designing algorithms for problems with least possible synchronization assumptions. Asynchrony in algorithms has been earnestly a desired property for algorithms to possess. Recently, Garg (SPAA, 2020) [6] showed that if lattices are induced among the state space, then an algorithm that traverses those lattices, under some additional constraints, guarantees convergence in asynchrony. In Section 2.2, we formally described the structure of such lattices. However, the problems studied in [6] are constrained to a class, which we call lattice-linear problems, and do not allow self-stabilization. In this chapter, we study whether there exist lattice-linear problems that allow self-stabilization.

Specifically, in this chapter, we study some parallel processing algorithms for multiplication and modulo operations. We demonstrate that the state transitions that are formed under these algorithms satisfy lattice-linearity, and these algorithms induce a lattice among the global states. Lattice-linearity implies that these algorithms can be implemented in asynchronous environments, where the nodes are allowed to read old information from each other. It means that these algorithms are guaranteed to converge correctly without any synchronization overhead. These algorithms also exhibit snap-stabilizing properties, i.e., starting from an arbitrary state, the sequence of state transitions made by the system strictly follows its specification.

The algorithms present in this chapter tolerate asynchrony in AMR model (cf. Section 2.1.1).

Organization of the Chapter

This chapter is organized as follows. In Section 3.1, we describe the definitions and notations that we specifically use in this chapter. We study lattice linearity of the multiplication operation in Section 3.2. Then, in Section 3.3, we study the lattice linearity of the modulo operation. Finally, we summarize the chapter in Section 3.5. We discuss the nature of the problems that are not naturally lattice-linear in Section 3.6; this section lays the foundation of the next chapter in which we study such problems and present algorithms for them.

3.1 Some Specific Preliminaries

This chapter focuses on multiplication and modulo operations, where the operands are n and m . In the computation $n \times m$ or $n \bmod m$, n and m are the values of these numbers respectively, and $|n|$ and $|m|$ are the length of the bitstrings required to represent n and m respectively.¹ If n is a bitstring, then $n[k]$ is the k^{th} bit of n (indices start from 1). For a bitstring n , $n[1]$ is the most significant bit of n and $n[|n|]$ is the least significant bit of n . We use $n[j : k]$ to denote the bitstring from j^{th} bit to k^{th} bit of n ; this includes

¹Since n and m are sequence of bit-values and if x is a sequence or a set, $|x|$ is used to denote the number of elements in x , $|n|$ and $|m|$ are the lengths of these bitstrings respectively. This notation does not represent the magnitude of their values.

$n[i]$ and $n[j]$. For simplicity, we stipulate that n and m are of lengths in some power of 2. Since size of n and m may be substantially different, we provide complexity results that are of the form $O(f(n, m))$ in all cases, where f is a function of n and m .

3.1.1 Additional Operations

We use the following string operations: (1) *append*(a, b), appends b to the end of a in $O(1)$ time, (2) *rshift*(a, k), deletes rightmost k bits of a in $O(k)$ time, and (3) *lshift*(a, k), appends k zeros to the right of a in $O(k)$ time.

$n \times m$ or $n \bmod m$ are typically thought of as arithmetic operations. However, when n and m are large, we view them as algorithms. In this chapter, we view them as parallel/distributed algorithms where the nodes collectively perform computations to converge to the final output.

In several places, we have used the functions $\text{MOD}(x, y)$, $\text{MUL}(x, y)$, and $\text{SUM}(x, y)$. These functions, respectively, compute $x \bmod y$, $x \times y$ and $x + y$.

3.1.2 Modulo: Some Classic Sequential Models

In this subsection, we discuss some sequential algorithms for computing $n \bmod m$. We will utilize these preliminary algorithms to analyze the effective time complexity of parallelized modulo operation. We consider two instances, one where both n and m are inputs and another where n is an input but m is hardcoded. We utilize these algorithms in Section 3.3.

The latter algorithm is motivated by algorithms such as RSA [13] where the value of n changes based on the message to be encrypted/decrypted, but the value of m is fixed once the keys are determined. Thus, some pre-processing can potentially improve the performance of the modulo operation; we observe that certain optimizations are possible. While the time and space complexities required for preprocessing in this algorithm are high, thereby making it impractical, it demonstrates a gap between the lower and upper bound in the complexity.

Modulo by Long Division

The standard long-division algorithm to compute $n \bmod m$ is shown in Figure 3.1.

Clearly, the number of iterations in this algorithm is bounded by $|n|$. In each of these iterations, the worst case complexity is $O(|m|)$ to perform the subtraction operation. Thus, the time complexity of standard long division is $O(|n| \times |m|)$ when m and n are both inputs to the algorithm.

Modulo by Constructing DFA

If the value of m is hardcoded in the algorithm, we use it to reduce the cost of the modulo operation by creating a deterministic finite automaton (DFA) $M = \langle Q, \Sigma, \delta, q_0 \rangle$, where (1) $Q = \{q_0..q_{m-1}\}$ is the set of all possible states of M , (2) $\Sigma = \{0, 1\}$ is the alphabet set, (3) δ is the transition function, the details of which we study in this section, and (4) q_0 is the initial state. If M has read the first k digits of n then

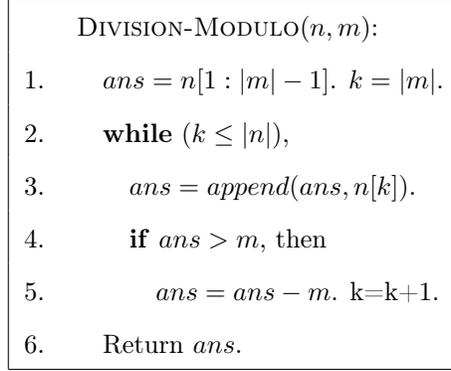


Figure 3.1: The standard long-division algorithm to compute $n \bmod m$.

its state would be $n[1 : k] \bmod m$. When M reads the next digit of n , $n[1 : (k + 1)] \bmod m$ is evaluated depending on the next digit. If the next digit is 0, the next state will be $(2 \times (n[1 : k] \bmod m)) \bmod m$, otherwise, it will be $(2 \times (n[1 : k] \bmod m) + 1) \bmod m$. Since the value of m is hardcoded, this DFA is assumed to be pre-computed. As an example, for $m = 3$, the corresponding DFA is provided in Example 3.1.

Example 3.1. A finite automaton M_3 computing $n \bmod 3$ ($m = 3$ is fixed) for any $n \in \mathbb{N}$ is shown in Figure 3.2. □

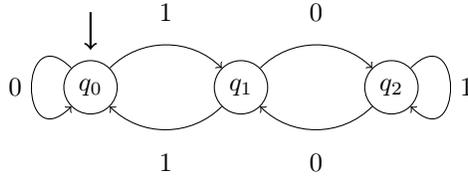


Figure 3.2: A finite automaton M_3 computing $n \bmod 3$ for any $n \in \mathbb{N}$.

With this DFA, the cost of computing the modulo operation corresponds to one DFA transition for each digit of n . Hence, the complexity of the corresponding operation is $O(|n|)$. M does not have any accepting states, which is unlike a usual finite automaton; the final state of M only tells us the value of the remainder which we would obtain after the computation of the expression $n \bmod m$.

We define the construction of the transition function δ of M in Figure 3.3. δ is constructed using the magnitude of m , and it is capable of reading a bitstring n starting from $n[1]$ and traversing through $n[|n|]$, reading every bit, sequentially, in each step. The problem is as follows: let that M has read the first z bits of n that evaluates to the value K (here, $K = n[1 : z]$), and let $K \bmod m = k$, so M is in state q_k . From here, we have to determine the next state based on whether the next bit M reads is 0 or 1, which would mean that the total value read will be $2 \times K + 0$ or $2 \times K + 1$.

We start by assigning edges from q_0 . $(2 \times 0 + 0) \bmod m$ is 0 and $(2 \times 0 + 1) \bmod m$ is 1, for example.

So $\delta(q_0, 0) = 0$ and $\delta(q_0, 1) = 1$. So we assign q_0 to transition to q_0 on input 0, and to q_1 on input 1. This method induces labelled edges between the states of M , such that those labels define which edge should be traversed based on what bit is read. After reading the first bit, assuming that M has read the value K so far and M is in state q_k , we assign the next state to be $(2 \times k + 0) \bmod m$ and $(2 \times k + 1) \bmod m$ for inputs 0 and 1 respectively. This is because $(2 \times K + 0) \bmod m$ and $(2 \times K + 1) \bmod m$ will be equal to $(2 \times k + 0) \bmod m$ and $(2 \times k + 1) \bmod m$ respectively. Note that if $2 \times k + 0$ (resp., $2 \times k + 1$) is greater than m , then we assign $(2 \times k + 0) \bmod m$ (resp., $(2 \times k + 1) \bmod m$) to be $(2 \times k + 0) - m$ (resp., $(2 \times k + 1) - m$) as $2 \times k + 0$ (and $2 \times k + 1$) cannot exceed $2 \times m$.

Elaborated definition of δ .

1. $v = 0$. $m_{rs} = rshift(m, 1)$.
 $b = \text{last bit of } m$. $i = 0$.
2. **for**(; $i < m_{rs}$; $i = i + 1$),
3. $\delta(q_i, 0) = q_v$. $v = v + 1$.
4. $\delta(q_i, 1) = q_v$. $v = v + 1$.
5. **if** $b = 1$, then
6. $\delta(q_i, 0) = q_v$. $\delta(q_i, 1) = q_0$. $v = 1$.
7. **else**, then
8. $\delta(q_i, 0) = q_0$. $\delta(q_i, 1) = q_1$. $v = 2$.
9. $i = i + 1$.
10. **for**(; $i < m$; $i = i + 1$),
11. $\delta(q_i, 0) = q_v$. $v = v + 1$.
12. $\delta(q_i, 1) = q_v$. $v = v + 1$.
13. $\delta(q_i, 0) = q_0$. $\delta(q_i, 1) = q_1$.

Figure 3.3: Definition of the transition function δ .

It can be clearly observed that while this approach takes $|n|$ steps, each step taking a constant amount of time, the time complexity (as well as the space complexity) of the required preprocessing is $O(m \times |m|)$, which is very high. Therefore, this approach is not practical when m is large. However, we consider it to observe that we obtain a time complexity of $O(|n|)$ to run this automaton to evaluate for the modulo operation if m can be hardcoded.

3.2 Parallelized Multiplication Operation

In this section, we demonstrate that the parallelized version of the standard multiplication algorithm as well as a parallelized version of Karatsuba's [7] algorithm presented in [8] meet the requirements of lattice-linearity, i.e. a system of nodes traverses a lattice of global states and provide the final output. We consider the problem where we want to compute $n \times m$.

3.2.1 Parallelizing Standard Multiplication

In this subsection, we present the parallelization of the standard multiplication algorithm. First, we discuss the key idea of the sequential algorithm, then we elaborate on the lattice-linearity of its parallelization.

Key Idea

In the standard multiplication, we multiply m with one digit of n at a time (for each digit of n), and then add all these multiplications, after left shifting those resultant strings appropriately. Suppose that we have two strings $a = n[1 : \lfloor n/2 \rfloor] \times m$ and $b = n[\lfloor n/2 \rfloor + 1 : n] \times m$. Then the resultant multiplication $n \times m$ will be equal to $lshift(a, |n| - \lceil n/2 \rceil) + b$.

Parallelization

This algorithm requires $2 \times |n| - 1$ nodes, and induces a binary tree among them. The root of the tree is marked as node 1 and any node i ($1 \leq i \leq |n| - 1$) has two children: node $2i$ and node $2i + 1$.

In this algorithm, every node stores two variables: *shift* and *ans*. We demonstrate that the computation of each of these variables is lattice-linear.

Computation of $i[shift]$: At the lowest level (level 1), the value of *shift* is set to 0. Consequently, at the next level (level 2), *shift* is set to 1. At all higher levels, *shift* of any node is computed to be twice the value of *shift* of its children, i.e., $i[shift]$ is set to $2 \times (2i)[shift]$. This can be viewed as a lattice-linear computation where a node is impedensable iff the following condition is satisfied. A impedensable node updates its value to be either 0 (at level 1), 1 (at level 2), or $2 \times (2i)[shift]$ (at level 3 and higher).

$$\text{IMPEDENSABLE-MULTIPLICATION-STANDARD-SHIFT}(i) \equiv \begin{cases} i[shift] \neq 0 & \text{if } i \geq |n| \\ i[shift] \neq 1 & \text{if } (2i)[shift] = (2i + 1)[shift] = 0 \\ i[shift] \neq 2 \times (2i)[shift] & \text{if } (2i)[shift] = (2i + 1)[shift] \geq 1 \end{cases}$$

Computation of $i[ans]$: The value of *ans* at the lowest level (level 1) is set to be the corresponding bit of n . At level 2, $i[ans]$ is computed to be equal to the bitstring stored in *ans* of left child (left-shifted by

$i[shift]$ bits (by 1 bit)) multiplied with m , added to the bitstring stored in ans of right child multiplied with m . At every level above level 2, $i[ans]$ is set by left shifting $(2i)[ans]$ by the $i[shift]$, and then adding $(2i + 1)[ans]$ to that value. Thus, to propagate the value of ans among the nodes correctly, we declare them to be impedensable as follows.

$$\text{IMPEDENSABLE-MULTIPLICATION-STANDARD-ANS}(i) \equiv \begin{cases} i[ans] \neq n[i - |n| + 1] & \text{if } i \geq |n|. \\ i[ans] \neq \text{lshift}(((2i)[ans] \times m), i[shift]) \\ \quad + ((2i + 1)[ans] \times m) & \text{if } (2i)[shift] = (2i + 1)[shift] = 0. \\ i[ans] \neq \text{lshift}((2i)[ans], i[shift]) + (2i + 1)[ans] & \text{if } (2i)[shift] = (2i + 1)[shift] \geq 1. \end{cases}$$

We observe that determining $\text{IMPEDENSABLE-MULTIPLICATION-STANDARD-ANS}(i)$ is more complex than determining $\text{IMPEDENSABLE-MULTIPLICATION-STANDARD-SHIFT}(i)$. However, we can eliminate it by observing that it suffices to update ans when $shift$ is updated. This requires that $i[shift]$ and $i[ans]$ are updated at a node i atomically in a single step. In that case, we can view the algorithm as Algorithm 3.1.

Algorithm 3.1. *Parallelized standard multiplication algorithm.*

Rules for node i .

$\text{IMPEDENSABLE-MULTIPLICATION-STANDARD-ANS}(i) \longrightarrow$

$$\begin{cases} i[shift] = 0, i[ans] = n[i - |n| + 1]. & \text{if } i \geq |n|. \\ i[shift] = 1, \\ \quad i[ans] = \text{lshift}(((2i)[ans] \times m), 1) + ((2i + 1)[ans] \times m). & \text{if } (2i)[shift] = 0. \\ i[shift] = 2 \times (2i)[shift], \\ \quad i[ans] = \text{lshift}((2i)[ans], i[shift]) + (2i + 1)[ans]. & \text{otherwise.} \end{cases}$$

From the above description, we see that the standard multiplication algorithm satisfies the constraints of lattice-linearity, which we prove in the following part of this subsection. This algorithm executes in $O(|m| \times \lg |n|)$ time. Its work complexity is $O(|n| \times |m|)$, which is same as the time complexity of the standard multiplication algorithm. The example below demonstrates the working of Algorithm 3.1.

Example 3.2. *In Figure 3.4, we demonstrate the multiplication of the bitstrings 00011011 (value of n) and 0101 (value of m) under Algorithm 3.1.* □

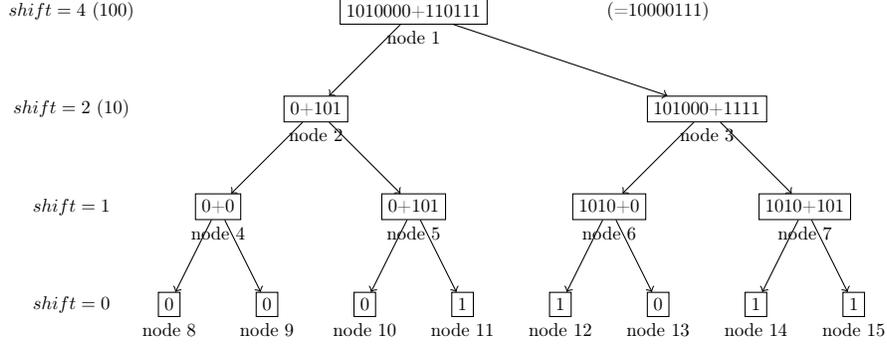


Figure 3.4: Multiplication of 00011011 and 0101 in base 2.

Lattice-Linearity

Lemma 3.1. *Given the input bitstrings n and m , the predicate*

$$\forall i : \neg \text{IMPEDENSABLE-MULTIPLICATION-STANDARD-ANS}(i)$$

is lattice-linear on $2|n| - 1$ computing nodes.

Proof. Let us assume that node 1 does not have the correct value of $1[ans] = n \times m$. This implies that (1) node 1 has a non-updated value in $1[ans]$ or $1[shift]$, in which case node 1 is impedensable, or (2) node 2 does not have the correct values $2[ans] = n[1 : |n|/2] \times m$ or $2[shift]$ or node 3 does not have the correct values $3[ans] = n[|n|/2 + 1 : |n|] \times m$ or $3[shift]$.

Recursively, this can be extended to any node i . Let that node i has stored an incorrect value in $i[ans]$ or $i[shift]$. If $i \leq |n| - 1$, then this means that (1) node i has a non-updated value in $i[ans]$ or $i[shift]$, in which case node i is impedensable, or (2) node $2i$ or node $2i + 1$ do not have the correct values in $(2i)[ans] = n[|n| - 2(|n| - 2i) + 1 : |n| - 2(|n| - 2i) + 2]$ or $(2i + 1)[ans] = n[|n| - 2(|n| - 2i - 1) + 1 : |n| - 2(|n| - 2i - 1) + 2]$ respectively. If $i \geq |n|$, then this implies that node i has not read the correct value $n[i - |n| + 1]$, in which case, again, node i is impedensable.

From these cases, we have that given a global state s , where $s = \langle \langle 1[ans], 1[shift] \rangle, \langle 2[ans], 2[shift] \rangle, \dots, \langle (2|n| - 1)[ans], (2|n| - 1)[shift] \rangle \rangle$, if s is impedensable, there is at least one node which is impedensable.

Next, we show that if some node is impedensable, then node 1 will not store the correct answer. $\forall i : i \in [1 : 2|n| - 1]$ node i is impedensable if it has a non-updated value in $i[ans]$ ($i[ans] = n[|n| - 2(|n| - i) + 1 : |n| - 2(|n| - i) + 2] \times m$ if $i \leq |n| - 1$ and $i[ans] = n[i - |n| + 1]$ if $i \geq |n|$) or $i[shift]$. This implies that the parent of node i will also store incorrect value in its ans or $shift$ variable. Recursively, we have that node 1 stores an incorrect value in $1[ans]$, and thus the global state is impedensable. \square

Lemma 3.2. *The predicate*

$$\forall i : \neg \text{IMPEDENSABLE-MULTIPLICATION-STANDARD-ANS}(i)$$

is a lattice-linear self-stabilizing predicate.

Proof. Since the lattice linearity was shown in Lemma 3.1, we only focus on the self-stabilizing aspect here. To show this, we need to show that this predicate is true in the supremum state.

We note that if s is the supremum of the induced lattice, then there is no outgoing edge from s to any other global state in the state transition graph. It means that in s , no node is enabled, and so, no node is impedensable. Thus, we have that the predicate

$$\forall i : \neg \text{IMPEDENSABLE-MULTIPLICATION-STANDARD-ANS}(i)$$

holds true in s . Since s is an arbitrary supremum, this predicate is a self-stabilizing predicate. \square

Theorem 3.1. *Algorithm 3.1 is silent and self-stabilizing.*

Proof. The nodes that have $\text{ID} \geq n$ (leaf nodes) read the bit-values directly from the input (cf. first rule of Algorithm 3.1), so their value is fixed immediately when they make their first move. Then, these nodes will not change their states. After that, recursively, all other nodes will update their state with respect to the state of their children (cf. second and third rules of Algorithm 3.1). If the nodes are arbitrarily initialized, then several nodes may need to update their state more than once.

This process will continue for all nodes in the tree, and the nodes will converge to a stable state bottom-to-top, in an acyclic fashion. Therefore, eventually, the root node will correct its own state. At this point, no node is enabled and the value of *ans* in the root node provides the answer. Thus, Algorithm 3.1 is silent and self-stabilizing. \square

3.2.2 Parallelized Karatsuba's Multiplication Operation

In this section, we study the lattice-linearity of the parallelization (of Karatsuba's [7] algorithm) that was presented in [8]. First we discuss the idea behind the sequential Karatsuba's algorithm, and then we elaborate on its lattice-linearity.

Key Idea of Sequential Karatsuba's Algorithm

The input is a pair of bitstrings n and m . This algorithm is recursive in nature. As the base case, when the length of n and m equals 1 then, the multiplication result is trivial. When the length is greater than 1, we let $n = \text{append}(a, b)$ and $m = \text{append}(c, d)$, where a and b are half the length of n , and c and d are half the length of m . Here, $\text{append}(a, b)$, for example, represents concatenation of a and b , which equals n .

Let $z = 2^{|b|}$. $n \times m$ can be computed as $a \times c \times z^2 + (a \times d + b \times c) \times z + b \times d$. $a \times d + b \times c$ can be computed as $(a + b) \times (c + d) - a \times c - b \times d$. Thus, to compute $n \times m$, it suffices to compute 3 multiplications

$a \times c$, $b \times d$ and $(a + b) \times (c + d)$. Hence, we can eliminate one of the multiplications. In the following section, we analyse the lattice-linearity of the parallelization of this algorithm as described in [8].

The CM Parallelization [8] for Karatsuba's Algorithm

The Karatsuba multiplication algorithm involves dividing the input string into substrings and use them to evaluate the multiplication recursively. In the parallel version of this algorithm, the recursive call is replaced by utilizing other (*children*) nodes to treat those substrings. We elaborate more on this in the following paragraphs. Consequently, this algorithm induces a tree among the computing nodes, where every non-leaf node has three children. This algorithm works in two phases, top-down and bottom-up. This algorithm uses four variables to represent the state of each node i : $i[n]$, $i[m]$, $i[ans]$ and $i[shift]$ respectively.

In the sequential Karatsuba's algorithm, both of the input strings n and m are divided into two substrings each, and the algorithm then runs recursively on three different input pairs computed from those excerpt bitstrings. In the parallel version, those recursive calls are replaced by *activating* three children nodes [8]. As a result of such parallelization, if there is no carry-forwarding due to addition, we require $\lg |n|$ levels, for which a total of $|n|^{\lg 3}$ nodes are required. However, if there is carry-forwarding due to additions, then we require $2 \lg |n|$ levels, for which a total of $|n|^{2 \lg 3}$ nodes are required.

In the top-down phase, if $|i[m]| > 1$ or $|i[n]| > 1$, then i writes (1) a and c to its left child, node $3i - 1$ ($(3i - 1)[m] = a$ and $(3i - 1)[n] = c$), (2) b and d to its middle child, node $3i$ ($(3i)[m] = b$ and $(3i)[n] = d$), and (3) $a + b$ and $c + d$ to its right child, node $3i + 1$ ($(3i + 1)[m] = a + b$ and $(3i + 1)[n] = c + d$). If $|i[m]| = |i[n]| = 1$, i.e., in the base case, the bottom-up phase begins and node i sets $i[ans] = i[m] \times i[n]$ that can be computed trivially since $|i[m]| = |i[n]| = 1$.

In the bottom-up phase, node i sets $i[ans] = (3i - 1)[ans] \times z^2 + ((3i + 1)[ans] - ((3i - 1)[ans] + (3i)[ans])) \times z + (3i)[ans]$. Notice that multiplication by z and z^2 corresponds to bit shifts and does not need an actual multiplication. Consequently, the product of $m \times n$ for node i is computed by this algorithm.

With some book-keeping (storing the place values of most significant bits of $a + b$ and $c + d$), a node i only needs to write the rightmost $\frac{|i[m]|}{2}$ and $\frac{|i[n]|}{2}$ bits to its children. Thus, we can safely assume that when a node writes m and n to any of its children, then m and n of that child are of equal length and are of length in some power of 2. (If we do not do the book-keeping, the required number of nodes increases, this number is upper bounded by $|n|^{2 \lg 3}$ as the number of levels is upper bounded by $2 \lg |n|$; this observation was not made in [8].) However, we do not show such book-keeping in the algorithm for brevity. Thus this algorithm would require $2 \lg |n|$ levels, i.e., $|n|^{2 \lg 3}$ nodes.

Computation of $i[shift]$: This algorithm utilizes *shift* to compute z . A node i updates $i[shift]$ by doubling the value of *shift* from its children. A node i evaluates that it is impedensable because of an

incorrect value of $i[shift]$ by evaluating the following macro.

$$\text{IMPEDENSABLE-MULTIPLICATION-KARATSUBA-SHIFT}(i) \equiv \left\{ \begin{array}{l} |i[m]| = 1 \wedge |i[n]| = 1 \wedge i[shift] \neq 0 \text{ OR} \\ (3i)[shift] = (3i-1)[shift] = 0 \leq (3i+1)[shift] \wedge i[shift] \neq 1 \text{ OR} \\ 0 < (3i)[shift] = (3i-1)[shift] \leq (3i+1)[shift] \wedge i[shift] \neq (3i)[shift] \times 2. \end{array} \right.$$

Computation of $i[m]$ and $i[n]$: To ensure that the data flows down correctly, we declare a node i to be impedensable as follows.

$$\text{IMPEDENSABLE-MULTIPLICATION-KARATSUBA-TOPDOWN}(i) \equiv \left\{ \begin{array}{l} i = 1 \wedge (i[m] \neq m \vee i[n] \neq n) \text{ OR} \\ ((|i[m]| > 1 \wedge |i[n]| > 1) \wedge \\ ((3i-1)[m] \neq i[m] \left[1 : \frac{|i[m]|}{2} \right] \text{ OR} \\ (3i-1)[n] \neq i[n] \left[1 : \frac{|i[n]|}{2} \right] \text{ OR} \\ (3i)[m] \neq i[m] \left[\frac{|i[m]|}{2} + 1 : |i[m]| \right] \text{ OR} \\ (3i)[n] \neq i[n] \left[\frac{|i[n]|}{2} + 1 : |i[n]| \right] \text{ OR} \\ (3i+1)[m] \neq i[m] \left[1 : \frac{|i[m]|}{2} \right] + i[m] \left[\frac{|i[m]|}{2} + 1 : |i[m]| \right] \text{ OR} \\ (3i+1)[n] \neq i[n] \left[1 : \frac{|i[n]|}{2} \right] + i[n] \left[\frac{|i[n]|}{2} + 1 : |i[n]| \right])). \end{array} \right.$$

Computation of $i[ans]$: To determine if a node i has stored $i[ans]$ incorrectly, it evaluates to be impedensable as follows.

$$\text{IMPEDENSABLE-MULTIPLICATION-KARATSUBA-BOTTOMUP}(i) \equiv \left\{ \begin{array}{l} |i[m]| = 1 \wedge |i[n]| = 1 \wedge i[ans] \neq i[m] \times i[n] \text{ OR} \\ |i[m]| > 1 \wedge |i[n]| > 1 \wedge (i[ans] \neq lshift((3i-1)[ans], i[shift]) \\ + lshift((3i+1)[ans] - (3i-1)[ans] - (3i)[ans], (3i)[shift]) \\ + (3i+1)[ans]) \end{array} \right.$$

Thus, Algorithm 3.2 is described as follows:

Algorithm 3.2. *Parallel processing version of Karatsuba's algorithm.*

Rules for node i .

IMPEDENSABLE-MULTIPLICATION-KARATSUBA-SHIFT(i) \rightarrow

$$\left\{ \begin{array}{ll} i[shift] = 0 & \text{if } |i[m]| = 1 \wedge |i[n]| = 1 \wedge i[shift] \neq 0. \\ i[shift] = 1 & \text{if } (3i)[shift] = (3i - 1)[shift] = 0 \\ & \leq (3i + 1)[shift] \wedge i[shift] \neq 1 \\ i[shift] = (3i)[shift] \times 2 & \text{otherwise} \end{array} \right.$$

IMPEDENSABLE-MULTIPLICATION-KARATSUBA-TOPDOWN(i) \rightarrow

$$\left\{ \begin{array}{ll} i[m] = m, i[n] = n & \text{if } i = 1 \wedge (i[m] \neq m \vee i[n] \neq n). \\ (3i - 1)[m] = i[m] \left[1 : \frac{|i[m]|}{2} \right] & \text{if } (3i - 1)[m] \neq i[m] \left[1 : \frac{|i[m]|}{2} \right]. \\ (3i - 1)[n] = i[n] \left[1 : \frac{|i[n]|}{2} \right] & \text{if } (3i - 1)[n] \neq i[n] \left[1 : \frac{|i[n]|}{2} \right]. \\ (3i)[m] = i[m] \left[\frac{|i[m]|}{2} + 1 : |i[m]| \right] & \text{if } (3i)[m] \neq i[m] \left[\frac{|i[m]|}{2} + 1 : |i[m]| \right]. \\ (3i)[n] = i[n] \left[\frac{|i[n]|}{2} + 1 : |i[n]| \right] & \text{if } (3i)[n] \neq i[n] \left[\frac{|i[n]|}{2} + 1 : |i[n]| \right]. \\ (3i + 1)[m] = i[m] \left[1 : \frac{|i[m]|}{2} \right] \\ \quad + i[m] \left[\frac{|i[m]|}{2} + 1 : |i[m]| \right] & \text{if } (3i + 1)[m] \neq i[m] \left[1 : \frac{|i[m]|}{2} \right]. \\ & \quad + i[m] \left[\frac{|i[m]|}{2} + 1 : |i[m]| \right]. \\ (3i + 1)[n] = i[n] \left[1 : \frac{|i[n]|}{2} \right] \\ \quad + i[n] \left[\frac{|i[n]|}{2} + 1 : |i[n]| \right] & \text{otherwise} \end{array} \right.$$

IMPEDENSABLE-MULTIPLICATION-KARATSUBA-BOTTOMUP(i) \rightarrow

$$\left\{ \begin{array}{ll} i[ans] = i[m] \times i[n] & \text{if } |i[m]| = 1 \wedge |i[n]| = 1 \\ i[ans] = lshift((3i - 1)[ans], i[shift]) + \\ \quad lshift((3i + 1)[ans] - (3i - 1)[ans] \\ \quad - (3i)[ans], (3i)[shift]) + (3i + 1)[ans] & \text{otherwise.} \end{array} \right.$$

Algorithm 3.2 converges in $O(|n|)$ time [8], and its work complexity is $O(n^{\lg 3})$, which is the time complexity of the sequential Karatsuba's algorithm [8].

Example 3.3. *Figure 3.5 evaluates 100×100 following Algorithm 3.2.*

□

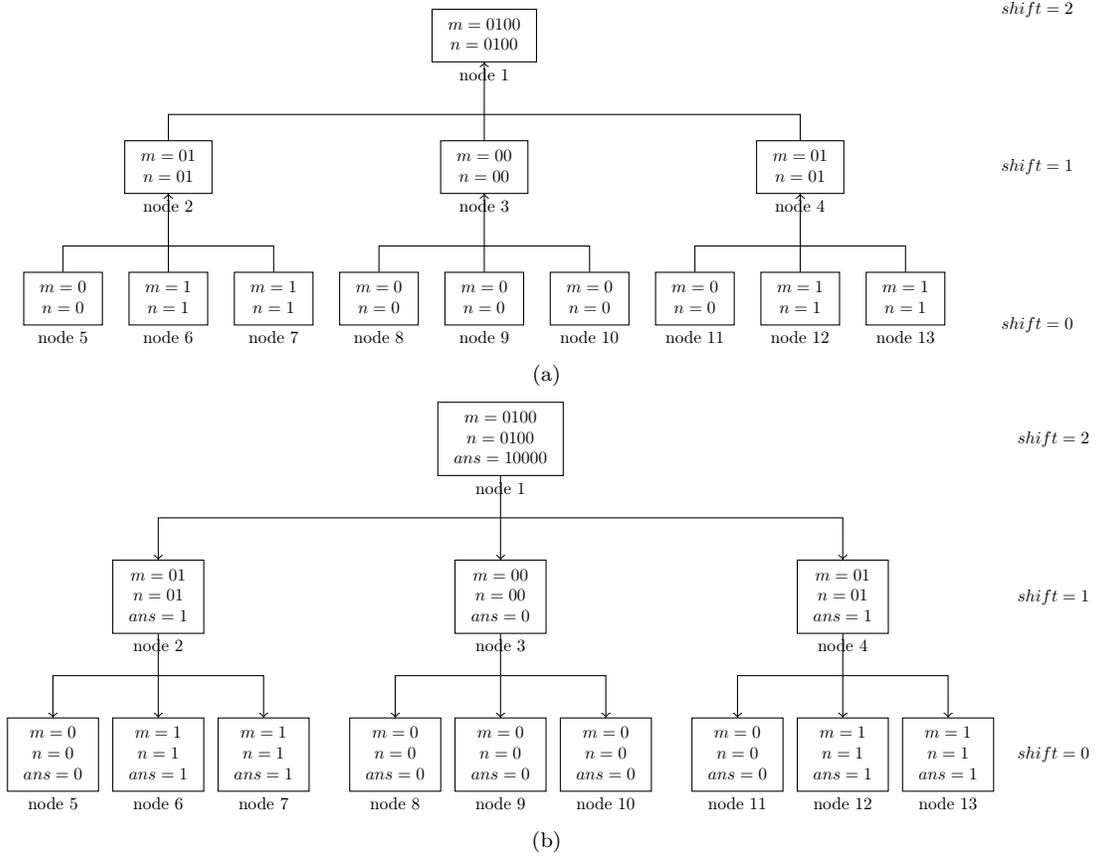


Figure 3.5: Demonstration of multiplication of 100 and 100 in base 2: (a) top down (b) bottom up.

Lattice-Linearity

Lemma 3.3. *Given the input bitstrings n and m , the predicate*

$$\forall i : \neg(\text{IMPEDENSABLE-MULTIPLICATION-KARATSUBA-SHIFT}(i) \vee \text{IMPEDENSABLE-MULTIPLICATION-KARATSUBA-TOPDOWN}(i) \vee \text{IMPEDENSABLE-MULTIPLICATION-KARATSUBA-BOTTOMUP}(i))$$

is lattice-linear on $|n|^{2 \lg 3}$ computing nodes.

Proof. For the global state to be optimal, in this problem, we require node 1 to store the correct multiplication result in $1[ans]$. To achieve this, each node i must have the correct value stored in $i[n]$ and $i[m]$, and their children must store correct values of n , m and ans according to the values of $i[n]$ and $i[m]$. This in turn requires all nodes to store the correct $i[shift]$ values.

Let us assume for contradiction that node 1 does not store the correct value in $1[ans]$ as $n \times m$. This implies that (1) node 1 does not have an updated value in $1[n]$ or $1[m]$, or (2) node 1 has a non-updated value of $1[ans]$, (3) node 1 has not written the updated values to $2[n]$ & $2[m]$, $3[n]$ & $3[m]$ or $4[n]$ & $4[m]$, (4)

node 1 has a non-updated value in $1[shift]$, or (5) nodes 2, 3 or 4 have incorrect values in their respective n , m , ans or $shift$ variables. In cases (1),..., (4), node 1 is impedensable.

Recursively, this can be extended to any node i . Let node i has stored an incorrect value in $i[ans]$ or $i[shift]$. Let $i > 1$. Then (1) node i has a non-updated value in $i[shift]$, $i[ans]$, $i[n]$ or $i[m]$, or (2) if $|i[m]| > 1$ or $|i[n]| > 1$, node i has not written updated values to $(3i-1)[n]$ & $(3i-1)[m]$ or $(3i)[n]$ & $(3i)[m]$ or $(3i+1)[n]$ & $(3i+1)[m]$, in which case node i is impedensable. In both these cases, node i is impedensable. It is also possible that at least one of the children of node i has incorrect values in its respective n , m , ans or $shift$ variables.

From these cases, we have that given a global state s , where $s = \langle \langle 1[n], 1[m], 1[ans] \rangle, \langle 2[n], 2[m], 2[ans] \rangle, \dots \rangle$, if s is impedensable, there is at least one node which is impedensable. This shows that if the global state is impedensable, then there exists some node i which is impedensable.

Next, we show that if some node is impedensable, then node 1 will not store the correct answer. Node 1 is impedensable if it has not read the correct value $1[m]$ and $1[n]$. Additionally, $\forall i : i \in [1 : n^{2 \lg 3}]$ node i is impedensable if (1) it has non-updated values in $i[ans]$ or $i[shift]$, (2) i has not written the correct values to $(3i-1)[n]$ & $(3i-1)[m]$ or $(3i)[n]$ & $(3i)[m]$ or $(3i+1)[n]$ & $(3i+1)[m]$. This implies that the parent of i will also store incorrect value in its ans or $shift$ variable. Recursively, we have that node 1 stores an incorrect value in $1[ans]$. Thus, the global state is impedensable. \square

With the arguments similar to those made in the proof of Lemma 3.2 and Theorem 3.1, we have the following.

Lemma 3.4. *The predicate*

$$\begin{aligned} \forall i : \neg (& \text{IMPEDENSABLE-MULTIPLICATION-KARATSUBA-SHIFT}(i) \vee \\ & \text{IMPEDENSABLE-MULTIPLICATION-KARATSUBA-TOPDOWN}(i) \vee \\ & \text{IMPEDENSABLE-MULTIPLICATION-KARATSUBA-BOTTOMUP}(i)) \end{aligned}$$

is a lattice-linear self-stabilizing predicate.

Theorem 3.2. *Algorithm 3.2 is silent and self-stabilizing.*

Remark: In Algorithm 3.2, an impedensable node updates the state of its children. We have done so for the brevity of the presentation of the algorithm. In practice, each child will notice that its local state does not tally with the state of its parent, and will then update its own state.

3.3 Parallel Processing Modulo Operation

In this section, we demonstrate parallel processing systems which can be used to compute a given modulo operation $n \bmod m$.

3.3.1 Using $|n|$ Processors

In this subsection, we discuss a lattice-linear method to compute modulo operation which requires $|n|$ processors. First, we discuss the key idea of the sequential algorithm, then we elaborate on the lattice-linearity of its parallelization.

Key Idea

This algorithm is based on the standard modulo operation. Suppose that we have computed $a = n[1 : |n| - 1] \bmod m$ and $b = n[|n|]$. Then we have that the resultant value of $n \bmod m$ is $(a \times 2 + b) \bmod m$, which is also equal to $(lshift(a, 1) + b) \bmod m$.

Parallelization

Every node, sequentially, reads a distinct bit of the input dividend n . Every node i will eventually store the value of $n[1 : i]$ under modulo m . The last node, indexed as node $|n|$, will store the final value, i.e. $n \bmod m$. We demonstrate two ways of executing this algorithm. One way is with using the machine M that we constructed in Section 3.1.2. Another way is to perform the computation without M where we use `DIVISION-MODULO()` that we defined in Section 3.1.2. We demonstrate these methods in the following.

Using M

In this part, we will utilize M to compute $n \bmod m$. Since every node i must store the value of $n[1 : i] \bmod m$, the impedensable node i can be defined as follows.

$$\text{IMPEDENSABLE-LINEAR-MODULO}(i) \equiv \begin{cases} i[ans] \neq n[1] & \text{if } i = 1 \\ (i[ans] \neq M((i-1)[ans], n[i])) & \text{otherwise} \end{cases}$$

In the definition of `IMPEDENSABLE-LINEAR-MODULO(i)`, $M(i, j)$ means that M is being invoked with an initial state q_i and an input $j \in \{0, 1\}$, i.e. $M(i, j) = \delta(q_i, j)$. If $\delta(q_i, j) = q_k$, then the execution of $M(i, j)$ will give k as output. The algorithm to compute $n \bmod m$ is demonstrated in Algorithm 3.3.

Algorithm 3.3. *Computing modulo on $|n|$ processors using M .*

Rules for node i .

$$\text{IMPEDENSABLE-LINEAR-MODULO}(i) \longrightarrow \begin{cases} i[ans] = n[1] & \text{if } i = 1. \\ i[ans] = M((i-1)[ans], n[i]) & \text{otherwise.} \end{cases}$$

The time complexity of this algorithm is $O(|n|)$. However, this method needs a preprocessing of $O(m \times$

$|m|$), which is quite high and impractical, especially if m is large. We present this result only to demonstrate that some pre-processing can reduce the complexity of the modulo operation substantially.

Theorem 3.3. *Given the input bitstrings n and m , the predicate*

$$\forall i : \neg \text{IMPEDENSABLE-LINEAR-MODULO}(i)$$

is lattice-linear on $|n|$ computing nodes.

Proof. Let us assume that node $|n|$ has incorrect value in $|n|[ans]$. This implies that (1) node $|n|$ does not have an updated value in $(|n|)[ans]$, in which case node $|n|$ is impedensable, or (2) node $|n| - 1$ has an incorrect value in $(|n| - 1)[ans]$.

Recursively, this can be extended to any node i . Let that node i has stored an incorrect value in $i[ans]$, then (1) node i has a non-updated value in $i[ans]$, in which case, node i is impedensable, or (2) node $i - 1$ has an incorrect value in $(i - 1)[ans]$. From these cases, we have that given a global state s , where $s = \langle 1[ans], 2[ans], \dots, |n|[ans] \rangle$, if s is impedensable, there is at least one node which is impedensable.

This shows that if the global state is impedensable, then there exists some node i which is impedensable.

Next, we show that if some node is impedensable, then node 1 will not store the correct answer. If node i is impedensable, then node i has a non-updated value in $i[ans]$. This implies that node $i + 1$ will also store incorrect value in $(i + 1)[ans]$. Recursively, we have that node $|n|$ stores an incorrect value in $|n|[ans]$, and thus the global state is impedensable. \square

With the arguments similar to those made in the proof of Lemma 3.2 and Theorem 3.1, we have the following.

Lemma 3.5. *The predicate*

$$\forall i : \neg \text{IMPEDENSABLE-LINEAR-MODULO}(i)$$

is a lattice-linear self-stabilizing predicate.

Theorem 3.4. *Algorithm 3.3 is silent and self-stabilizing.*

Using Long Division

If we utilize DIVISION-MODULO() instead of M in Algorithm 3.3 (and, subtraction in place of M in the definition of IMPEDENSABLE-LINEAR-MODULO(i)), then every node takes $O(|m|)$ time because of the subtraction in DIVISION-MODULO(), which implies that the total work complexity is $O(|n| \times |m|)$. Node i will compute the value of $n[1 : i] \bmod m$ by the end of time-step t . Therefore, the time complexity of this algorithm is $O(|n| \times |m|)$ to compute $n \bmod m$, which is the same as the work complexity of this algorithm.

Discussion

The behaviour of these methods is lattice-linear, but similar to a uniprocessor computation, in the sense that if we ran these methods on a uniprocessor machine, then it will take the same order of time. In Section 3.3.2, we present algorithms which exploit the power of a distributed system better.

3.3.2 Using $4 \lfloor n/m \rfloor - 1$ Processors

In this section, we present a parallel processing algorithm to compute $n \bmod m$ using $4 \times \lfloor n/m \rfloor - 1$ computing nodes. First, we discuss the key idea of the sequential algorithm, then we elaborate on the lattice-linearity of its parallelization.

Key Idea

This algorithm better parallizes the idea discussed in Section 3.3.1. Suppose that we have computed $a = n[1 : \lfloor n/2 \rfloor] \bmod m$ and $b = n[\lfloor n/2 \rfloor + 1 : n]$. Then the resultant value of $n \bmod m$ is $(a \times 2^{\lceil n/2 \rceil} + b) \bmod m$, which is also equal to $(lshift(a, \lceil n/2 \rceil) + b) \bmod m$.

Parallelization

The algorithm induces a binary tree among the nodes based on their ids; there are $2 \times \lfloor n/m \rfloor$ nodes in the lowest level (level 1). This algorithm starts from the leaves where all leaves compute and store, in sequence, a substring of n of length $\lfloor m/2 \rfloor$ under modulo m . In the induced binary tree, the computed modulo result by sibling nodes at level ℓ is sent to the parent. Consecutively, those parents at level $\ell + 1$, contiguously, store a larger substring of n (double the bits that each of their children covers) under modulo m . We elaborate this procedure in this subsection. This algorithm uses three variables to represent the state of each node i : $i[shift]$, $i[pow]$ and $i[ans]$.

Computation of $i[shift]$: The variable $shift$ stores the required power of 2. At any node at level 1, $shift$ is 0. At level 2, the value of $shift$ at any node is $\lfloor m/2 \rfloor$. At any higher level, the value of $shift$ is twice the value of shift of its children. IMPEDENSABLE-LOG-MODULO-SHIFT, in this context, is defined below.

$$\text{IMPEDENSABLE-LOG-MODULO-SHIFT}(i) \equiv \begin{cases} i[shift] \neq 0 & \text{if } i \geq 2 \times \lfloor n/m \rfloor \\ i[shift] \neq \lfloor m/2 \rfloor & \text{if } (2i)[shift] = (2i + 1)[shift] = 0 \\ i[shift] \neq 2 \times (2i)[shift] & \text{if } (2i)[shift] = (2i + 1)[shift] \geq \lfloor m/2 \rfloor \end{cases}$$

Computation of $i[pow]$: The goal of this computation is to set $i[pow]$ to be $2^{i[shift]} \bmod m$, whenever the level of i is greater than 1. This can be implemented using the following definition for IMPEDENSABLE-LOG-MODULO-POW.

$$\text{IMPEDENSABLE-LOG-MODULO-POW}(i) \equiv \begin{cases} i[\text{pow}] \neq 1 & \text{if } i[\text{shift}] = 0 \\ i[\text{pow}] \neq 2^{\lfloor \frac{|m|}{2} \rfloor} & \text{if } i[\text{shift}] = |m|/2 \\ i[\text{pow}] \neq ((2i)[\text{pow}])^2 \pmod m & \text{otherwise} \end{cases}$$

By definition, $i[\text{pow}]$ is less than m . Also, computation of pow requires multiplication of two numbers that are upper-bounded by $|m|$. Hence, this computation can benefit from parallelization of Algorithm 3.2. However, as we will see later, the complexity of this algorithm (for modulo) is dominated by the modulo operation happening in individual nodes which is $O(|m|^2)$, we can use the sequential version of Karatsuba's algorithm for multiplication, without affecting the order of the time complexity of this algorithm.

Computation of $i[\text{ans}]$: We split n into strings of size $\lfloor \frac{|m|}{2} \rfloor$, the number representing this substring is less than m . At the lowest level (level 1), $i[\text{ans}]$ is set to the corresponding substring. At higher levels, $i[\text{ans}]$ is set to $(i[\text{pow}] \times (2i)[\text{ans}] + (2i + 1)[\text{ans}]) \pmod m$. This computation also involves multiplication of two numbers whose size is upper bounded by $|m|$. An impedensable node i from a non-updated $i[\text{ans}]$ can be evaluated using $\text{IMPEDENSABLE-LOG-MODULO-ANS}(i)$.

$$\text{IMPEDENSABLE-LOG-MODULO-ANS}(i) \equiv \begin{cases} i[\text{ans}] \neq n[(i - 2 \times \lfloor \frac{|n|}{|m|} \rfloor) \times \lfloor \frac{|m|}{2} \rfloor + 1 : (i - 2 \times \lfloor \frac{|n|}{|m|} \rfloor + 1) \times \lfloor \frac{|m|}{2} \rfloor] & \text{if } i[\text{shift}] = 0 \\ i[\text{ans}] \neq \text{MOD}(\text{SUM}(\text{MUL}((2i)[\text{ans}], i[\text{pow}]), (2i + 1)[\text{ans}], m)) & \text{otherwise} \end{cases}$$

We describe the algorithm as Algorithm 3.4.

Algorithm 3.4. *Modulo computation by inducing a tree among the nodes.*

Rules for node i .

IMPEDENSABLE-LOG-MODULO-SHIFT(i) \rightarrow

$$\begin{cases} i[shift] = 0 & \text{if } i \geq 2 \times |n|/|m| \\ i[shift] = \frac{|m|}{2} & \text{if } (2i)[shift] = (2i+1)[shift] = 0 \\ i[shift] = 2 \times (2i)[shift] & \text{if } (2i)[shift] = (2i+1)[shift] \geq |m|/2 \end{cases}$$

IMPEDENSABLE-LOG-MODULO-POW(i) \rightarrow

$$\begin{cases} i[pow] = 1 & \text{if } i[shift] = 0 \\ i[pow] = 2^{\frac{|m|}{2}} & \text{if } i[shift] = |m|/2 \\ i[pow] = \text{MOD}(\text{MUL}(i[pow], i[pow]), m) & \text{otherwise} \end{cases}$$

IMPEDENSABLE-LOG-MODULO-ANS(i) \rightarrow

$$\begin{cases} i[ans] = n[(i - 2 \times \frac{|n|}{|m|}) \times \frac{|m|}{2} + 1 : (i - 2 \times \frac{|n|}{|m|} + 1) \times \frac{|m|}{2}] & \text{if } i[shift] = 0 \\ i[ans] = \text{MOD}(\text{SUM}(\text{MUL}((2i)[ans], i[pow]), (2i+1)[ans]), m) & \text{otherwise} \end{cases}$$

Example 3.4. *Figure 3.6 shows the computation of $11011 \bmod 11$ as performed by Algorithm 3.4.* □

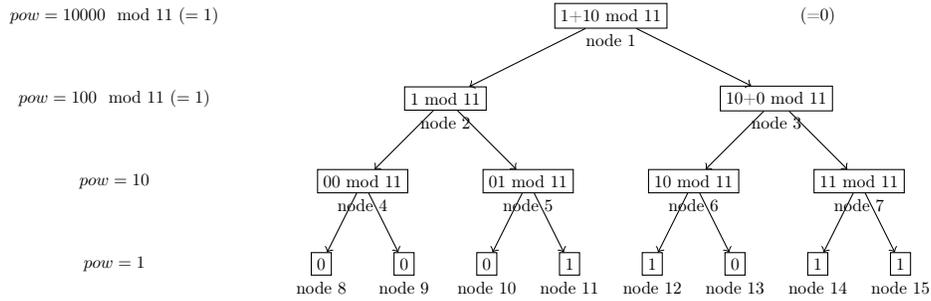


Figure 3.6: Processing $11011 \bmod 11$ following Algorithm 3.4.

Lattice-Linearity

Theorem 3.5. *Given the input bitstrings n and m , the predicate*

$$\begin{aligned} \forall i : & \neg(\text{IMPEDENSABLE-LOG-MODULO-SHIFT}(i) \vee \\ & \text{IMPEDENSABLE-LOG-MODULO-POW}(i) \vee \\ & \text{IMPEDENSABLE-LOG-MODULO-ANS}(i)) \end{aligned}$$

is lattice-linear on $4|n|/|m| - 1$ computing nodes.

Proof. For the global state to be optimal, under this algorithm, we require node 1 to store the correct modulo result in $1[ans]$. To achieve this, each node i must have the correct value of $i[ans]$. This in turn requires all nodes to store the correct $i[shift]$ and $i[pow]$ values.

Let us assume for contradiction that node 1 has incorrect value in $1[ans]$. This implies that (1) node 1 has a non-updated value in $i[shift]$ or $i[pow]$, or (2) node 1 does not have an updated value in $i[ans]$. In both these cases, node 1 is impedensable. It is also possible that node 2 or node 3 have an incorrect value in their variables.

Recursively, this can be extended to any node i . Let node i has stored an incorrect value in $i[ans]$. If $i < 2(\lfloor n/m \rfloor)$, then (1) node i has a non-updated value in $i[ans]$, $i[pow]$ or $i[shift]$, in which case node i is impedensable or (2) node $2i$ or node $2i + 1$ have incorrect value in their respective $shift$, pow or ans variables. If $i \geq 2(\lfloor n/m \rfloor)$, then i does not have values $i[shift] = 0$, $i[pow] = 1$ or a correct $i[ans]$ value, in which case i is impedensable. From these cases, we have that given a global state s , where $s = \langle \langle 1[shift], 1[pow], 1[ans] \rangle, \langle 2[shift], 2[pow], 2[ans] \rangle, \dots, \langle (4\lfloor n/m \rfloor - 1)[shift], (4\lfloor n/m \rfloor - 1)[pow], (4\lfloor n/m \rfloor - 1)[ans] \rangle \rangle$, if s is impedensable, there is at least one node which is impedensable.

This shows that if the global state is impedensable, then there exists some node i which is impedensable.

Next, we show that if some node is impedensable, then node 1 will not store the correct answer. $\forall i : i \in [1 : 4\lfloor n/m \rfloor - 1]$ node i is impedensable if it has non-updated values in $i[ans]$, $i[pow]$ or $i[shift]$. This implies that the parent of node i also stores incorrect value in its ans variable. Recursively, we have that node 1 stores an incorrect value in $1[ans]$, and thus the global state is impedensable. \square

With the arguments similar to those made in the proof of Lemma 3.2 and Theorem 3.1, we have the following.

Lemma 3.6. *The predicate*

$$\begin{aligned} \forall i : \neg (&IMPEDENSABLE-LOG-MODULO-SHIFT(i) \vee \\ &IMPEDENSABLE-LOG-MODULO-POW(i) \vee \\ &IMPEDENSABLE-LOG-MODULO-ANS(i)) \end{aligned}$$

is a lattice-linear self-stabilizing predicate.

Theorem 3.6. *Algorithm 3.4 is silent and self-stabilizing.*

Time Complexity Analysis

Algorithm 3.4 is a general algorithm that uses the $MOD(MUL(\dots))$ and $MOD(SUM(\dots))$. For some given x, y and z values, $MOD(MUL(x, y), z)$ (resp., $MOD(SUM(x, y), z)$) involves first the multiplication (resp., addition) of two input values x and y and then evaluating the resulting value under modulo z . These functions

can be implemented in different ways. Choices for these implementations affect the time complexity. We consider the following approaches.

Modulo via Long Division

First, we consider the standard approach for computing $\text{MOD}(\text{MUL}(\dots))$ and $\text{MOD}(\text{SUM}(\dots))$. Observe that in Algorithm 3.4, if we compute $\text{MOD}(\text{MUL}(x, y))$ then $x, y < m$. Hence, we can use Karatsuba's parallelized algorithm from Section 3.2, where both the input numbers are less than m . Using the analysis from Section 3.2, we have that each multiplication operation has a time complexity of $O(|m|)$.

Subsequently, to compute the mod operation, we need to compute $xy \bmod m$ where xy is upto $2|m|$ digits long. Using the standard approach of long division, we will need $|m|$ iterations where in each iteration, we need to do a subtraction operation with numbers that are $|m|$ digits long. Hence, the complexity of this approach is $O(|m|^2)$ per modulo operation. Since this complexity is higher than the cost of multiplication, the overall time complexity is $O(|m|^2 \times \lg \frac{|n|}{|m|})$.

Modulo by Using M

The previous approach used m and n as inputs. Next, we consider the case where m is hardcoded in the algorithm. As discussed in Section 3.1.2, we observe that these problems occur in practice. Our analysis is intended to provide lower bounds on the complexity of the modulo operation when m is hardcoded. Similar to Section 3.1.2, the pre-processing required in these algorithms makes them impractical in practice. However, we present them to show that there is a potential to reduce the complexity by some pre-processing.

We can use Algorithm 3.2 for multiplication; each multiplication operation will have a time complexity of $O(|m|)$. Subsequently, to compute the mod operation, we need to compute $xy \bmod m$ where xy is upto $2|m|$ digits long. Using M , we will need $2m$ iterations; each iteration takes a constant amount of time. Hence, the complexity of this approach is $O(|m|)$ per modulo operation. Since this complexity is higher than the cost of multiplication, the overall time complexity is $O(|m| \times \lg \frac{|n|}{|m|})$.

Modulo by Constructing Transition Functions

In this part, we again consider the case where m is hardcoded. If m is fixed, we can create a table δ_{sum} of size $m \times m$ where an entry at location (i, j) represents $i + j \bmod m$ in $O(m^2)$ time. Using δ_{sum} , we can create another transition function δ_{mul} of size $m \times m$ where an entry at location (i, j) represents $i \times j \bmod m$ in $O(m^2)$ time.

Using a preprocessed δ_{mul} , the time complexity of a $\text{MOD}(\text{MUL}(\dots))$ operation becomes $O(1)$. As a result, the overall complexity of the modulo operation becomes $O(\lg \frac{|n|}{|m|})$. The pre-processing required in this method also is high. However, the effective time complexity of the modulo operation is reduced even more, as compared to the method that uses M , which is discussed above.

3.4 Discussion on Common Properties of These Algorithms

In this section, we look at some common properties that are present in the problems and algorithms discussed in the preceding sections. Effectively, we also provide an alternate visualization to the abstraction of the lattices induced by the algorithms present in this chapter.

3.4.1 Data Dependency Among Nodes

In Section 3.2.2, for example, we showed how Algorithm 3.2 is lattice-linear by showing that given any suboptimal global state, we can point out specific nodes that are impedensable. Any impedensable node i has only one choice of action, which implies that a total order is induced among all the local states that i can visit. Such a total order, induced among the local states of every node, gives rise to the induction of a lattice among the global states.

Let that *source* of the variable $i[var]$ in a node i be the node that i depends on to evaluate $i[var]$. For example, under Algorithm 3.2, node i depends on node $3i - 1$, node $3i$ and node $3i + 1$ to evaluate $i[ans]$. Thus the source nodes for node i with respect to the evaluation of $i[ans]$ are node $3i - 1$, node $3i$ and node $3i + 1$. Similarly the source node of i with respect to $i[m]$ or $i[n]$ is node $\lfloor \frac{i+1}{3} \rfloor$.

Let that $SOURCE(i, var)$ is the set of nodes that are the source of i with respect to var . Thus, under Algorithm 3.2, for example, $SOURCE(i, ans) = \{3i - 1, 3i, 3i + 1\}$. $SOURCE(i, m) = \left\{ \left\lfloor \frac{i+1}{3} \right\rfloor \right\}$. However, $SOURCE(1, m) = \phi$ because i is receiving m as part of the input. Similarly, for all other algorithms, we can define the source nodes for all the nodes with respect to any given variable.

Let $VARIABLES(i)$ be the set of the names of all variables of node i . We use a macro $DEPENDS(i)$; a recursive definition for this macro is present in Figure 3.7.

$$\begin{array}{l} \text{DEPENDS}(i) \supseteq \bigcup_{var \in \text{VARIABLES}(i)} \text{SOURCE}(i, var). \\ \text{DEPENDS}(i) \supseteq \bigcup_{j \in \text{DEPENDS}(i), var \in \text{VARIABLES}(j)} \text{SOURCE}(j, var). \end{array}$$

Figure 3.7: Definition of the macro `depends`.

3.4.2 Induction of \prec -lattice

In Algorithm 3.2, for m and n , the information of m and n for i is set based on the values of the parent of i . Hence, the $DEPENDS(i)$ will contain all the ancestor nodes of i in the tree. In addition, the ans variable of i is based on the children of i . Hence $DEPENDS(i)$ will (also) contain the descendants of i in the tree. For example, in Figure 3.5, $DEPENDS(3) = \{1, 8, 9, 10\}$ and $DEPENDS(8) = \{1, 3\}$.

Let $IS-BAD(i, var)$ be true if and only if i is impedensable with respect to some variable var , i.e., based on the values of the variables in $SOURCE(i, var)$, i has not computed var correctly yet. We define state value of a node i in a global state s as follows. All the macros are also computed in the same global state s .

$$\begin{aligned} \text{STATE-VALUE}(i, s) = & \\ & |\{var \mid var \in \text{VARIABLES}(i) : \text{IS-BAD}(i, var)\}| \\ & + |\{var \mid var \in \text{VARIABLES}(j) : j \in \text{DEPENDS}(i, var) : \text{IS-BAD}(j, var)\}| \end{aligned}$$

We define the rank of a global state s as follows.

$$\text{RANK}(s) = \sum_{\text{each node } i} \text{STATE-VALUE}(i, s).$$

From the perspective of, for example, Algorithm 3.2, a total order is induced among the local state visited by a node; $\text{STATE-VALUE}(i)$ describes the badness of the local state of a node i , which decreases monotonously as the nodes execute under Algorithm 3.2. Similarly, for all other algorithms, a total order is defined similarly using $\text{STATE-VALUE}(i)$.

As a consequence of the total order that is defined by $\text{STATE-VALUE}(i)$, a lattice among the global states can be observed with respect to the rank of the system; if the rank of a state s is nonzero, then there is some node i that is impedensable in s . Let that only node i changes its state and as a consequence, s transitions to state s' . Then, we have that $s \prec s'$ where $s[i] \prec s'[i]$. This forms a \prec -lattice among the global states where $s[i] \prec s'[i]$ iff $\text{STATE-VALUE}(i, s) > \text{STATE-VALUE}(i, s')$ and $s \prec s'$ iff $\text{RANK}(s) > \text{RANK}(s')$. Rank is 0 at the supremum of the lattice, which is the optimal state.

From the above observation, we have that the system is able to converge from an arbitrary state to the required state within the expected number of time steps. This allows providing new inputs to a parallel processing system without needing to refresh variables of the nodes.

A problem is lattice-linear if it can be modelled in such a way that an impedensable node must change its state in order for the system to reach the optimal state [6]. From the above discussion, we have the following theorem about multiplication and modulo operations.

3.5 Summary of the Chapter

The contribution of this chapter is two-fold, one is applicative and the other is mathematical. First, we show that the parallelization of multiplication and modulo is lattice-linear. Due to lattice-linearity, we have that the algorithms we study in this chapter are tolerant to asynchrony. Second, we show two different distributive lattice structures, for both multiplication and modulo, which guarantee convergence in asynchronous environments. This chapter is the first work that shows that a lattice-linear problem can be solved under two different lattice structures. Specifically, considering (any) one of these problems in both the lattice structures, (1) the numbers of nodes are different, so the size of the global states in both the lattice structures is different, and (2) the numbers of children that a node has are different.

Multiplication and Modulo are among the fundamental mathematical operations. Fast parallel processing

algorithms for such operations reduce the execution time of the applications which they are employed in. In this chapter, we showed that these problems are lattice-linear. In this context, we studied parallelization of the standard multiplication and a parallelization of Karatsuba’s algorithm. In addition, we studied parallel processing algorithms for the modulo operation.

The presence of lattice-linearity in problems and algorithms allows nodes to execute asynchronously. This is specifically valuable in parallel algorithms where synchronization can be removed as is. These algorithms are snap-stabilizing, which means that the state transitions of the system strictly follow its specification. They are also self-stabilizing, i.e., the supremum states in the lattices induced under the respective predicates are the optimal states.

Utilizing these algorithms, the available cluster or GPU power can be used to compute the multiplication and modulo operations on big-number inputs. In this case, a synchronization primitive also does not need to be deployed. Also, the circuit does not need to be refreshed before providing it with a new input. This is also very fruitful, for example, in Karatsuba’s multiplication the time that it would take to refresh the circuit is $O(|n|^{2\lg 3})$, but even without refreshing, we obtain the final answer in $O(n)$ time. This shows the gravity of the utility of the self-stabilizing property of these algorithms. Thus a plethora of applications will benefit from the observations presented in this chapter.

3.6 Non-Lattice-Linear Problems

Unlike the problems that we studied in this chapter, certain problems are *non-lattice-linear problems*. In those problems, given a suboptimal global state, the problem does not stipulate, from a specific set of nodes, to change their state. In such problems, there are instances in which the impedensable nodes cannot be determined naturally, i.e., in those instances $\exists s : \neg \mathcal{P}(s) \wedge (\forall i : \exists s' : \mathcal{P}(s') \wedge s[i] = s'[i])$. For such problems, \prec -lattices may be induced algorithmically, through *lattice-linear algorithms*.

In Chapter 4 (and 5), we study non-lattice linear problems. Minimal dominating set, minimal vertex cover and maximal independent set are examples of non-lattice linear problems: in such problems for any subject node i , an optimal global state can be reached without changing the local state of i .

CHAPTER 4

EVENTUALLY LATTICE-LINEAR ALGORITHMS

In Chapter 3, we study examples of lattice-linear problems and present self-stabilizing algorithms that guarantee convergence in asynchrony. These algorithms are capable of converging in asynchrony because they exploit the property of lattice-linearity of the problems that they are developed for.

In this chapter, we study whether lattice-linearity can be extended for problems that are not lattice-linear. This is one of the issues that were pointed out in [6]: whether non-lattice-linear problems can be solved under the model of lattice-linearity. The behaviour of non-lattice-linear problems makes it seem impossible since the nature of these problems does not provide the definition of impedensable nodes. However, we find that a total order can be induced among the local states of nodes algorithmically, even if the problem does not define how impedensable nodes can be identified.

In this chapter, we introduce the class of *eventually lattice-linear algorithms*. We present eventually lattice-linear self-stabilizing algorithms for service demand based minimal dominating set (SDMDS), minimal vertex cover (MVC), maximal independent set (MIS), graph colouring (GC) and 2-dominating set (2DS) problems.

Eventually lattice-linear algorithms induce lattices in a subset of the state space. These algorithms first (1) guarantee that from any arbitrary state, the system reaches a state in one of the induced lattices, and then (2) these algorithms behave lattice-linearly, and make the system traverse that lattice and reach an optimal state.

We proceed as follows. We begin with the SDMDS problem which is a generalization of the minimal dominating set. We devise a self-stabilizing algorithm for SDMDS. We scrutinize this algorithm and decompose it into two parts, the second of which satisfies the lattice-linearity property of [6] if it begins in a *feasible* state. Furthermore, the first part of the algorithm ensures that the algorithm reaches a *feasible* state. We show that the resulting algorithm is self-stabilizing, and the algorithm has *limited-interference* property (discussed in Section 4.2.5) due to which it is tolerant to the nodes reading old values of other nodes. We also demonstrate that this approach is generic. It applies to various other problems including MVC MIS, GC and 2DS.

The algorithms for SDMDS, MVC and MIS converge in 1 round plus n moves, the algorithm for GC converges in $n + 4m$ moves, and the algorithm for 2DS converges in 1 round plus $2n$ moves. Adding to the fact that these algorithms do not require a synchronous environment to execute, these results are an improvement over the algorithms present in the literature.

We also present some experimental results that show the efficacy of eventually lattice-linear algorithms

in real-time shared memory systems. Specifically, we compare our algorithm for MIS (Algorithm 4.3) with algorithms presented in [14] and [15]. The experiments are conducted in cuda environment, which is built on shared memory model.

The algorithms present in this chapter tolerate asynchrony in AMR model (cf. Section 2.1.1).

Organization of the Chapter

This chapter is organized as follows. In Section 4.1, we describe the algorithm for the service demand based minimal dominating set problem. In Section 4.2, we analyze the characteristics of that algorithm and show that it is eventually lattice-linear. We use the structure of eventually lattice-linear self-stabilizing algorithms to develop algorithms for minimal vertex cover, maximal independent set, graph colouring and 2-dominating set problems, respectively, in Sections 4.3, 4.4, 4.5 and 4.6. Then, in Section 5.6, we compare the convergence speed of the algorithm presented in Section 4.4 with other algorithms (for the maximal independent set problem) in the literature (specifically, [14] and [15]). Finally, we summarize the chapter in Section 4.8.

4.1 Service Demand based Minimal Dominating Set

In this section, we introduce a generalization of the minimal dominating set (MDS) problem (Section 4.1.1), the service demand based minimal dominating set (SDMDS) problem, and describe an algorithm to solve it (Section 4.1.2).

4.1.1 Problem Description

The SDMDS problem, a generalization of MDS, is a simulation, on an arbitrary graph G , in which all nodes have some demands to be fulfilled and they offer some services. If a node i is in the dominating set then it can not only serve all its own demands D_i , but also offer services from, its set of services S_i , to its neighbours. If i is not in the dominating set, then it is considered dominated only if each of its demands in D_i is being served by at least one of its neighbours that is in the dominating set.

Definition 4.1. *Service demand based minimal dominating set problem (SDMDS).* In the service demand based minimal dominating set problem, the input is a graph G and a set of services S_i and a set of demands D_i for each node i in G ; the task is to compute a minimal set \mathcal{D} such that for each node i ,

1. either $i \in \mathcal{D}$, or
2. for each demand d in D_i , there exists at least one node j in Adj_i such that $d \in S_j$ and $j \in \mathcal{D}$.

In the above generalization of the MDS problem, if all nodes have same set X as their services and demands, i.e., $\forall i : S_i = X$ and $D_i = X$, then it is equivalent to MDS.

In the following subsection, we present a self-stabilizing algorithm for the minimal SDMDS problem. Each node i is associated with variable $i[st]$ with domain $\{IN, OUT\}$. $i[st]$ defines the state of i . We define

\mathcal{D} to be the set $\{i \in V(G) : i[st] = IN\}$.

4.1.2 Algorithm for SDMDS Problem

The list of constants, provided with the input, is in Table 4.1.

| Constant | What it stands for |
|----------|--|
| D_i | the set of demands of node i . |
| S_i | the set of services provided by node i . |

Table 4.1: Constants provided with the input.

The macros that we utilize are described in Table 4.2. Recall that \mathcal{D} is the set of nodes which currently have the state as IN . A node i is *addable* if there is at least one demand of i that is not being serviced by any neighbour of i that is in \mathcal{D} . A node i is *removable* if $\mathcal{D} \setminus \{i\}$ is also a dominating set given that \mathcal{D} is a dominating set. The *dominators* of i are the nodes that are (possibly) dominating node i : if some node j is in $\text{DOMINATORS-OF}(i)$, then there is at least one demand $d \in D_i$ such that $d \in S_j$. i is *impedensable* if i is removable and there is no node k that is removable and is of an ID higher than i , such that k and i are able to serve for some common node j .

| |
|--|
| $\mathcal{D} \equiv \{i \in V(G) : i[st] = IN\}$. |
| $\text{ADDABLE-SDMDS-ELL}(i) \equiv i[st] = OUT \wedge$ $(\exists d \in D_i, \forall j \in Adj_i : d \notin S_j \vee j[st] = OUT)$. |
| $\text{REMOVABLE-SDMDS-ELL}(i) \equiv (\forall d \in D_i : (\exists j \in Adj_i : d \in S_j \wedge j[st] = IN)) \wedge$ $(\forall j \in Adj_i, \forall d \in D_j : d \in S_i \Rightarrow$ $(\exists k \in Adj_j, k \neq i : (d \in S_k \wedge k[st] = IN)))$. |
| $\text{DOMINATORS-OF}(i) \equiv$ $\{j \in Adj_i, j[st] = IN : \exists d \in D_i : d \in S_j\} \cup \{i\}$ if $i[st] = IN$ $\{j \in Adj_i, j[st] = IN : \exists d \in D_i : d \in S_j\}$ otherwise. |
| $\text{IMPEDENSABLE-SDMDS-ELL}(i) \equiv i[st] = IN \wedge \text{REMOVABLE-SDMDS-ELL}(i) \wedge$ $(\forall j \in Adj_i, \forall d \in D_j : d \in S_i \Rightarrow$ $((\forall k \in \text{DOMINATORS-OF}(j), k \neq i : (d \in S_k \wedge k[st] = IN)) \Rightarrow$ $(k[id] < i[id] \vee \neg \text{REMOVABLE-SDMDS-ELL}(k))))$. |

Table 4.2: Macros used in the algorithm for SDMDS problem.

The general idea our algorithm is as follows.

1. A node enters the dominating set unconditionally if it is addable. This ensures that G enters a state

where the set of nodes in \mathcal{D} form a (possibly non-minimal) dominating set. If \mathcal{D} is a dominating set, we say that the corresponding state is a *feasible* state.

2. While entering the dominating set is not lattice-linear, the instruction governing the leaving of the dominating set is lattice-linear. Node i leaves the dominating set iff it is impedensable. Specifically, if i serves for a demand d in D_j where $j \in Adj_i$ and the same demand is also served by another node k ($k \in Adj_j$) then i leaves only if (1) $k[id] < i[id]$ or (2) k is not removable. This ensures that if some demand d of D_j is satisfied by both i and k both of them cannot leave the dominating set simultaneously. This ensures that j will remain dominated.

Thus, the rules for Algorithm 4.1 are as follows:

Algorithm 4.1. *Rules for node i .*

| |
|---|
| $ADDABLE\text{-}SDMDS\text{-}ELL(i) \longrightarrow i[st] = IN.$ $IMPEDENSABLE\text{-}SDMDS\text{-}ELL(i) \longrightarrow i[st] = OUT.$ |
|---|

We decompose Algorithm 4.1 into two parts: (1) Algorithm 4.1.1, that only consists of first guard and action of Algorithm 4.1 and (2) Algorithm 4.1.2, that only consists of the second guard and action of Algorithm 4.1. We use this decomposition in the following section of this chapter to relate the algorithm to eventual lattice-linearity.

4.2 Lattice-Linear Characteristics of the Algorithm for SDMDS

In this section, we analyze the characteristics of Algorithm 4.1 to demonstrate that it is eventually lattice-linear. We proceed as follows. In Section 4.2.1, we state the propositions which define the feasible and optimal states of the SDMDS problem, along with some other definitions. In Section 4.2.2, we show that G reaches a state where it manifests a (possibly non-minimal) dominating set. In Section 4.2.3, we show that after when G reaches a feasible state, Algorithm 4.1 behaves like a lattice-linear algorithm. In Section 4.2.4, we show that when \mathcal{D} is a minimal dominating set, no nodes are enabled. In Section 4.2.5, we argue that because there is a bound on interference between Algorithm 4.1.1 and Algorithm 4.1.2 even when the nodes read old values, Algorithm 4.1 is an eventually lattice-linear self-stabilizing (ELLSS) algorithm. In Section 4.2.6, we study the time and space complexity attributes of Algorithm 4.1.

4.2.1 Propositions Stipulated by the SDMDS Problem

The SDMDS problem stipulates that the nodes whose state is IN must collectively form a dominating set. We represent this proposition as $\mathcal{P}'_{sdm ds}$.

$$\mathcal{P}'_{sdm ds}(\mathcal{D}) \equiv \forall i \in V(G) : (i \in \mathcal{D} \vee (\forall d \in D_i, \exists j \in Adj_i : (d \in S_j \wedge j \in \mathcal{D}))).$$

The SDMDS problem stipulates an additional condition that \mathcal{D} should be a minimal dominating set. We represent this proposition as $\mathcal{P}_{sdm ds}$.

$$\mathcal{P}_{sdmds}(\mathcal{D}) \equiv \mathcal{P}'_{sdmds}(\mathcal{D}) \wedge (\forall i \in \mathcal{D}, \neg \mathcal{P}'_{sdmds}(\mathcal{D} \setminus \{i\})).$$

If $\mathcal{P}'_{sdmds}(\mathcal{D})$ is true, then G is in a *feasible* state. And, if $\mathcal{P}_{sdmds}(\mathcal{D})$ is true, then G is in an *optimal* state.

Based on the above definitions, we define two scores with respect to the global state, RANK and BADNESS. RANK determines the number of nodes needed to be added to \mathcal{D} to change \mathcal{D} to a dominating set. BADNESS determines the number of nodes that are needed to be removed from \mathcal{D} to make it a minimal dominating set, given that \mathcal{D} is a (possibly non-minimal) dominating set.

Definition 4.2. $RANK(\mathcal{D}) \equiv \min\{|\mathcal{D}'| - |\mathcal{D}| : \mathcal{P}'_d(\mathcal{D}') \wedge \mathcal{D} \subseteq \mathcal{D}'\}$.

Definition 4.3. $BADNESS(\mathcal{D}) \equiv \max\{|\mathcal{D}| - |\mathcal{D}'| : \mathcal{P}'_d(\mathcal{D}') \wedge \mathcal{D}' \subseteq \mathcal{D}\}$.

4.2.2 Guarantee to Reach a Feasible State by Algorithm 4.1.1

We show that under Algorithm 4.1.1, G is guaranteed to reach a feasible state.

Lemma 4.1. *Let $t.\mathcal{D}$ be the value of \mathcal{D} at the beginning of round t . If $t.\mathcal{D}$ is not a dominating set then $(t+1).\mathcal{D}$ is a dominating set.*

Proof. Let i be a node such that $i \in t.\mathcal{D}$ and $i \notin (t+1).\mathcal{D}$, i.e., i leaves the dominating set in round t . This means that i remains dominated and all nodes in Adj_i remain dominated, even when i is removed. This implies that i will not reduce the feasibility of $t.\mathcal{D}$; it will not increase the value of RANK.

Now let ℓ be a node such that $\ell \notin t.\mathcal{D}$ which is addable when it evaluates its guards in round t . This implies that $\exists d \in D_\ell$ such that d is not present in S_j for any $j \in Adj_\ell$ that is in the dominating set. According to the algorithm, the guard of the second action is true for ℓ . This implies that $\ell[st]$ will be set to IN .

It can also be possible for the node ℓ that it is not addable when it evaluates its guards in round t . This may happen if some other nodes around ℓ already decided to move to \mathcal{D} , and as a result ℓ is now dominated. Hence $\ell \notin (t+1).\mathcal{D}$ and we have that ℓ is dominated at round $t+1$.

Therefore, we have that $(t+1).\mathcal{D}$ is a dominating set, which may or may not be minimal. \square

From Lemma 4.1, we have that if at the beginning of some round, G is in a state where $RANK > 0$, then by the end of that round, RANK will be 0.

4.2.3 Lattice-Linearity of Algorithm 4.1.2

In the following lemma, we show that Algorithm 4.1.2 is lattice-linear.

Lemma 4.2. *If $t.\mathcal{D}$ is a non-minimal dominating set then under Algorithm 4.1 (more specifically, Algorithm 4.1.2), there exists at least one node such that G cannot reach a minimal dominating set until that node is removed from the dominating set.*

Proof. Since \mathcal{D} is a dominating set, the first guard is false for all nodes in G .

Since \mathcal{D} is not minimal, there exists at least one node that must be removed in order to make \mathcal{D} minimal. Let S' be the set of nodes which are removable. Let x be some node in S' . If x is not serving any node, then IMPEDENSABLE-SDMDS-ELL(x) is trivially true. Otherwise there exists at least one node j which is served by x , that is, $\exists d \in D_j : d \in S_x$. We study two cases which are as follows: (1) for some node j served by x , there does not exist another node $b \in S'$ which serves j , and (2) for any node $b \in S'$ such that x and b serve some common node j , $b[id] < M[id]$.

In the first case, x cannot be removed because IMPEDENSABLE-SDMDS-ELL(x) is false and, hence, x cannot be in S' , thereby leading to a contradiction. In the second case, IMPEDENSABLE-SDMDS-ELL(x) is true and IMPEDENSABLE-SDMDS-ELL(b) is false since $b[id] < M[id]$. Thus, node b cannot leave the dominating set until x leaves. In both the cases, we have that j stays dominated.

Since ID of every node is distinct, we have that there exists at least one node x for which IMPEDENSABLE-SDMDS-ELL(x) is true. For example, IMPEDENSABLE-SDMDS-ELL is true for the node with the highest ID in S' ; G cannot reach a minimal dominating set until x is removed from the dominating set. \square

From Lemma 4.2, it follows that Algorithm 4.1.2 satisfies the condition of lattice-linearity as described in Section 2.3. It follows that if we start from a state where \mathcal{D} is a (possibly non-minimal) dominating set and execute Algorithm 4.1.2 then it will reach a state where \mathcal{D} is a minimal dominating set even if nodes are executing with old information about others. Next, we have the following result which follows from Lemma 4.2.

Lemma 4.3. *Let $t.\mathcal{D}$ be the value of \mathcal{D} at the beginning of round t . If $t.\mathcal{D}$ is a non-minimal dominating set then $|(t+1).\mathcal{D}| \leq |t.\mathcal{D}| - 1$, and $(t+1).\mathcal{D}$ is a dominating set.*

Proof. From Lemma 4.2, at least one node x (including the maximum ID node in S' from the proof of Lemma 4.2) would be removed in round t . Furthermore, since \mathcal{D} is a dominating set, ADDABLE-SDMDS-ELL(i) is false at every node i . Thus, no node is added to \mathcal{D} in round t . Thus, the $|(t+1).\mathcal{D}| \leq |t.\mathcal{D}| - 1$.

For any node x that is removable, IMPEDENSABLE-SDMDS-ELL(i) is true only if any node j which is (possibly) served by x has other neighbours (of a lower ID) which serve the demands which x is serving to it. This guarantees that j stays dominated and hence $(t+1).\mathcal{D}$ is a dominating set. \square

4.2.4 Termination of Algorithm 4.1

The following lemma studies the action of Algorithm 4.1 when \mathcal{D} is a minimal dominating set.

Lemma 4.4. *Let $t.\mathcal{D}$ be the value of \mathcal{D} at the beginning of round t . If \mathcal{D} is a minimal dominating set, then $(t+1).\mathcal{D} = t.\mathcal{D}$.*

Proof. Since \mathcal{D} is a dominating set, $\text{ADDABLE-SDMDS-ELL}(i)$ is false for every node in $V(G)$, i.e., the first action is disabled for every node in $V(G)$. Since \mathcal{D} is minimal, $\text{IMPEDENSABLE-SDMDS-ELL}(i)$ is false for every node i in \mathcal{D} . Hence, the second action is disabled at every node i in \mathcal{D} . Thus, \mathcal{D} remains unchanged. \square

4.2.5 Eventual Lattice-Linearity of Algorithm 4.1

Lemma 4.2 showed that Algorithm 4.1.2 is lattice-linear. In this subsection, we make additional observations about Algorithm 4.1 to generalize the notion of lattice-linearity to eventually lattice-linear algorithms. We have the following observations.

1. From Lemma 4.1, starting from any state, Algorithm 4.1 will reach a feasible state even if a node reads old information about the neighbours. This is due to the fact that Algorithm 4.1.1 only adds nodes to \mathcal{D} . If incorrect information about the state of neighbours causes i not to be added to \mathcal{D} , this will be corrected when i executes again and obtains recent information about neighbours. If incorrect information causes i to be added to \mathcal{D} unnecessarily, it does not affect this claim.
2. From Lemma 4.2, if we start G in a feasible state where no node has incorrect information about the neighbours in the initial state then Algorithm 4.1.2 reaches a minimal dominating set. Note that this claim remains valid even if the nodes execute actions of Algorithm 4.1.2 with old information about the neighbours as long as the initial information they use is correct.
3. We observe that Algorithm 4.1.1 and Algorithm 4.1.2 have very limited interference with each other, and so an arbitrary graph G will reach an optimal state even if nodes are using old information.

From the above observations, if we allow the nodes to read old values, then the nodes can violate the feasibility of G finitely many times and so G will eventually reach a feasible state and stay there forever. We introduce the class of eventually lattice-linear algorithms (ELLA). Algorithm 4.1 is an ELLA.

Definition 4.4. *Eventually Lattice-Linear Algorithms (ELLA).* *An algorithm A is ELLA for a problem P , represented by a predicate \mathcal{P} , if its rules can be split into two sets of rules F_1 and F_2 and there exists a subset S_f of the state space S , such that*

- (a) *Any computation of A (from its permitted initial states) eventually reaches a state where S_f is stable in A , i.e., S_f is true and remains true subsequently.*
- (b) *Rules in F_1 are disabled in a state in S_f .*
- (c) *F_2 is a lattice-linear algorithm, i.e., it induces a total order among the local states visited by the nodes, given that the system initializes in a state in S_f .*

Definition 4.5. *Eventually Lattice-Linear Self-Stabilizing (ELLSS) Algorithms.* *Continuing from Definition 4.4, A is an ELLSS algorithm iff F_1 takes the system to a state in S_f from an arbitrary state,*

and F_2 is capable of taking the system from any state in S_f to an optimal state.

Remark: The algorithms that we study in this chapter are ELLSS algorithms, i.e., they follow Definition 4.5.

Notice that Algorithm 4.1 is an ELLSS algorithm.

In Algorithm 4.1, F_1 corresponds to Algorithm 4.1.1 and F_2 corresponds to Algorithm 4.1.2. This algorithm satisfies the properties of Definition 4.5.

Example 4.1. We illustrate the eventual lattice-linear structure of Algorithm 4.1 where we consider the special case where all nodes have the same single service and demand. Effectively, it becomes a case of minimal dominating set.

In Figure 4.1, we consider an example of graph G_4 containing four nodes connected in such a way that they form two disjoint edges, i.e., $V(G_4) = \{v_1, v_2, v_3, v_4\}$ and $E(G_4) = \{\{v_1, v_2\}, \{v_3, v_4\}\}$.

We write a state of this graph as $(v_1[st], v_2[st], v_3[st], v_4[st])$. As shown in this figure, of the 16 states in the state space, 9 are part of 4 disjoint lattices. These are feasible states, i.e., states where nodes with st equals IN form a (possibly non-minimal) dominating set. And, the remaining 7 are not part of any lattice. These are infeasible states, i.e., states where nodes with st equals IN do not form a dominating set. The states not taking part in any lattice structure (the infeasible states) are not shown in Figure 4.1.

In a non-feasible state, some node will be addable. The instruction executed by addable nodes is not lattice-linear: an addable node moves in the dominating set unconditionally. After this, when no node is addable, then the global state s becomes feasible state, i.e., s manifests a valid dominating set. In s , however, some nodes may be removable. Only the removable nodes can be impedensable. The instruction executed by an impedensable node is lattice-linear.

E.g., notice in Figure 4.1 (a), assuming that the initial state is (IN, IN, IN, IN) , that v_2 and v_4 are impedensable. Since they execute asynchronously, a lattice is induced among all possible global states that G_4 transitions through. If only v_2 (respectively, v_4) executes, the global state we obtain is (IN, OUT, IN, IN) (respectively, (IN, IN, IN, OUT)). Since eventually both the nodes change their local states, we obtain the global state (IN, OUT, IN, OUT) . \square

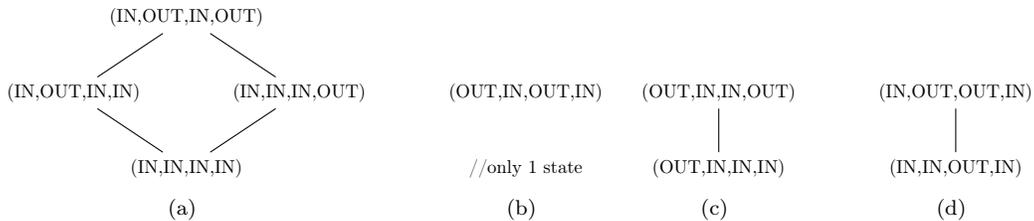


Figure 4.1: Example lattice induced by Algorithm 4.1.1 in G_4 (G_4 is described in Example 4.1).

4.2.6 Analysis of Algorithm 4.1: Time and Space complexity

Theorem 4.1. *Starting from an arbitrary state, Algorithm 4.1 reaches an optimal state within $2n$ moves (or more precisely 1 round plus n moves).*

Proof. From Lemma 4.1, we have that starting from an arbitrary state, Algorithm 4.1 will reach a feasible state within one round (or within n moves).

After that, if the input graph G is not in an optimal state, then at least one node moves out such that G stays in a feasible state (Lemma 4.3). Thus, G manifests an optimal state within n additional moves. \square

Corollary 4.1. *Algorithm 4.1 is self-stabilizing and silent.*

Observation 4.1. *At any time-step, a node will take $O((\Delta)^4 \times (\max_d)^2)$ time, where (1) Δ is the maximum degree of any node in $V(G)$, and (2) \max_d is the total number of distinct demands made by all the nodes in $V(G)$.*

4.3 Applying ELLSS in Minimal Vertex Cover

The execution of Algorithm 4.1 was divided in two phases, (1) where the system reaches a feasible state (reduction of RANK to 0), and (2) where the system reaches an optimal state (reduction of BADNESS to 0).

Such design defines the concept of ELLSS algorithms. This design can be extended to numerous other problems where the optimal global state can be defined in terms of a minimal (or maximal) set \mathcal{S} of nodes. This includes the minimal vertex cover (MVC) problem, maximal independent set problem and their variants. In this section, we discuss the extension to MVC.

Definition 4.6. Minimal Vertex Cover. *In the MVC problem, the input is an arbitrary graph G , and the task is to compute a minimal set \mathcal{V} such that for any edge $\{i, j\} \in E(G)$, $(i \in \mathcal{V}) \vee (j \in \mathcal{V})$. If a node i is in \mathcal{V} , then $i[st] = IN$, otherwise $i[st] = OUT$.*

The proposition \mathcal{P}'_v defining a feasible state and the proposition \mathcal{P}_v defining the optimal state can be defined as follows.

$$\begin{aligned} \mathcal{P}'_v(\mathcal{V}) &\equiv \forall i \in V(G) : ((i \in \mathcal{V}) \vee (\forall j \in Adj_i, j \in \mathcal{V})). \\ \mathcal{P}_v(\mathcal{V}) &\equiv \mathcal{P}'_v(\mathcal{V}) \wedge (\forall i \in \mathcal{V}, \neg \mathcal{P}'_v(\mathcal{V} \setminus \{i\})). \end{aligned}$$

To develop an algorithm for MVC, we utilize the macros in Table 4.3. A node i is *removable* if all the nodes in its neighbourhood are in the vertex cover (VC). i is *addable* if i is not in the VC and there is some node adjacent to it that is not in the VC. i is *impedensable* if i is in the VC, and i is the highest ID node that is removable in its distance-1 neighbourhood.

Based on the definitions above, the algorithm for MVC is described as follows. If a node is addable, then it moves into the VC. If a node is impedensable, then it moves out of the VC.

| |
|--|
| $\text{REMOVABLE-MVC-ELL}(i) \equiv \forall j \in \text{Adj}_i, j[\text{st}] = \text{IN}.$ |
| $\text{ADDABLE-MVC-ELL}(i) \equiv i[\text{st}] = \text{OUT} \wedge (\exists j \in \text{Adj}_i : j[\text{st}] = \text{OUT}).$ |
| $\text{IMPEDENSABLE-MVC-ELL}(i) \equiv i[\text{st}] = \text{IN} \wedge \text{REMOVABLE-MVC-ELL}(i) \wedge$ $(\forall j \in \text{Adj}_i : j[\text{id}] < i[\text{id}] \vee \neg \text{REMOVABLE-MVC-ELL}(j)).$ |

Table 4.3: Macros used in the algorithm for MVC.

Algorithm 4.2. *Rules for node i .*

| |
|---|
| $\text{ADDABLE-MVC-ELL}(i) \longrightarrow i[\text{st}] = \text{IN}.$ |
| $\text{IMPEDENSABLE-MVC-ELL}(i) \longrightarrow i[\text{st}] = \text{OUT}.$ |

Algorithm 4.2 is an ELLSS algorithm in that it satisfies the conditions in Definition 4.5, where F_1 corresponds to the first action of Algorithm 4.2, F_2 corresponds to its second action, and S_f is the set of the states for which \mathcal{P}'_v holds true. Thus, starting from any arbitrary state, the algorithm eventually reaches a state where \mathcal{V} is a minimal vertex cover.

Lemma 4.5. *Algorithm 4.2 is a silent eventually lattice-linear self-stabilizing algorithm for minimal vertex cover.*

Proof. In an arbitrary non-feasible state (where the input graph G does not manifest a valid VC), there is at least one node that is addable. An addable node immediately executes the first instruction of Algorithm 4.2 and moves in the VC. This implies that by the end of the first round, we obtain a valid (possibly non-minimal) VC.

If the input graph G is in a feasible, but not optimal, state (where G manifests a non-minimal VC), then there is at least one removable node. This implies that there is at least one impedensable node i in that state (e.g., the removable node with the highest ID). Under Algorithm 4.2, any node in Adj_i will not execute until i changes its state. i is removable because all nodes in Adj_i , along with i , are in the vertex cover. Thus i must execute so that it becomes non-removable. This shows that the second rule in Algorithm 4.2 is lattice-linear.

In a non-minimal, but valid, VC, there is at least one node that is impedensable, thus, with every move, the size of the vertex cover, manifested by G , reduces by 1. Also, notice that when an impedensable node i changes its state, no node in Adj_i changes its state simultaneously. Thus, the validity of the vertex cover is not impacted when i moves. Therefore, Algorithm 4.2 is self-stabilizing.

When G manifests a minimal vertex cover, no node is addable or removable. This shows that Algorithm 4.2 is silent. □

Observe that in Algorithm 4.2, the definition of IMPEDENSABLE relies only on the information about distance-2 neighbours. Hence, the evaluation of guards take $O(\Delta^2)$ time. In contrast, (the standard) minimal dominating set problem would require the information of distance-4 neighbours to evaluate IMPEDENSABLE. Hence, the evaluation of guards in that would take $O(\Delta^4)$ time. This algorithm converges in $2n$ moves (or more precisely 1 round plus n moves).

4.4 Applying ELLSS in Maximal Independent Set

In this section, we consider the application of ELLSS in the problem of maximal independent set (MIS). Unlike MVC and SDMDS problems where we tried to reach a minimal set, here, we have to obtain a maximal set.

Definition 4.7. Maximal Independent Set. *In the maximal independent set (MIS) problem, the input is an arbitrary graph G , and the task is to compute a maximal set \mathcal{I} such that for any two nodes $i \in \mathcal{I}$ and $j \in \mathcal{I}$, if $i \neq j$, then $\{i, j\} \neq E(G)$.*

The proposition \mathcal{P}'_i defining a feasible state and the proposition \mathcal{P}_i defining the optimal state can be defined as follows.

$$\begin{aligned}\mathcal{P}'_i(\mathcal{I}) &\equiv \forall i \in V(G) : ((i \notin \mathcal{I}) \vee (\forall j \in Adj_i : j \notin \mathcal{I})). \\ \mathcal{P}_i(\mathcal{I}) &\equiv \mathcal{P}'_i(\mathcal{I}) \wedge (\forall i \in V(G) : \neg \mathcal{P}'_i(\mathcal{I} \cup \{i\})).\end{aligned}$$

To develop the algorithm for MIS, we define the macros in Table 4.4. A node i is *addable* if all the neighbours of i are out of the independent set (IS). A node is *removable* if i is in the IS and there is some neighbour of i that is also in IS. i is *impedensable* if i is out of the IS, and i is the highest ID node in its distance-1 neighbourhood that is addable.

| |
|--|
| $\text{ADDABLE-MIS-ELL}(i) \equiv \forall j \in Adj_i, j[st] = OUT.$ $\text{REMOVABLE-MIS-ELL}(i) \equiv i[st] = IN \wedge (\exists j \in Adj_i : j[st] = IN).$ $\text{IMPEDENSABLE-MIS-ELL}(i) \equiv i[st] = OUT \wedge \text{ADDABLE-MIS-ELL}(i) \wedge$ $(\forall j \in Adj_i : j[id] < i[id] \vee \neg \text{ADDABLE-MIS-ELL}(j)).$ |
|--|

Table 4.4: Macros used in the algorithm for MIS.

Based on the definitions above, the algorithm for MIS is described as follows. If a node i is impedensable, then it moves into the IS. If i is removable, then it moves out of the IS.

Algorithm 4.3. *Rules for node i .*

| |
|---|
| $\text{REMOVABLE-MIS-ELL}(i) \longrightarrow i[st] = OUT.$ $\text{IMPEDENSABLE-MIS-ELL}(i) \longrightarrow i[st] = IN.$ |
|---|

This algorithm is an ELLSS algorithm as well: as per Definition 4.5, F_1 corresponds to the first action of Algorithm 4.2, F_2 corresponds to its second action, and S_f is the set of the states for which \mathcal{P}'_i holds true. Thus, starting from any arbitrary state, the algorithm eventually reaches a state where \mathcal{I} is a maximal independent set.

Lemma 4.6. *Algorithm 4.3 is a silent eventually lattice-linear self-stabilizing algorithm for maximal independent set.*

Proof. In an arbitrary non-feasible state (where the input graph G does not manifest a valid IS), there is at least one node that is removable. A removable node immediately executes the first instruction of Algorithm 4.3 and moves out of the IS. This implies that by the end of the first round, we obtain a valid (possibly non-minimal) IS.

If the input graph G is in a feasible, but not optimal, state (where G manifests a non-minimal IS), then there is at least one addable node. This implies that there is at least one impedensable node i in that state (e.g., the addable node with the highest ID). Under Algorithm 4.2, any node in Adj_i will not execute until i changes its state. i is addable because all nodes in Adj_i , along with i , are out of the independent set. Thus i must execute so that it becomes non-addable. This shows that the second rule in Algorithm 4.3 is lattice-linear.

Since in a non-minimal, but valid, independent set, there is at least one node that is impedensable, we have that with every move, the size of the independent set, manifested by G , reduces by 1. Also, notice that when an impedensable node i changes its state, no node in Adj_i changes its state simultaneously. Thus, the validity of the independent set is not impacted when i moves. Therefore, we have that Algorithm 4.3 is self-stabilizing.

When G manifests a maximal independent set, no node is removable or addable. This shows that Algorithm 4.3 is silent. □

In Algorithm 4.3, the definition of ADDABLE relies only on the information about distance-2 neighbours. Hence, the evaluation of guards take $O(\Delta^2)$ time. This algorithm converges in $2n$ moves (or more precisely 1 round plus n moves).

4.5 Applying ELLSS in Colouring

In this section, we extend ELLSS algorithms to graph colouring. In the *graph colouring* (GC) problem, the input is a graph G and the task is to (re-)assign colours to all the nodes such that no two adjacent nodes have the same colour.

Definition 4.8. Graph colouring. *In the GC problem, the input is an arbitrary graph G with some initial colouring assignment $\forall i \in V(G) : i[\text{colour}] \in \mathbb{N}$. The task is to (re)assign the colour values of the nodes such*

that any adjacent nodes should not have a conflict (i.e., should not have the same colour), and there should not be a node whose colour can be reduced without conflict.

Unlike MVC, MDS or MIS, colouring does not have a binary domain. Instead, we correspond the equivalence of changing the state to *IN* to the case where a node sets its colour to $i[id] + n$. And, the equivalence of changing the state to *OUT* corresponds to the case where a node decreases its colour.

The proposition \mathcal{P}'_c defining a feasible state and the proposition \mathcal{P}_c defining an optimal state is defined below. \mathcal{P}_c is true when all the nodes have lowest available colour, that is, for any node i and for all colours c in $[1 : i[colour] - 1]$, c equals the colour of one of the neighbours j of i .

$$\begin{aligned}\mathcal{P}'_c(G) &\equiv \forall i \in V(G), \forall j \in Adj_i : i[colour] \neq j[colour]. \\ \mathcal{P}_c(G) &\equiv \mathcal{P}'_c \wedge (\forall i \in V(G) : (\forall c \in [1 : i[colour] - 1] : (\exists j \in Adj_i : j[colour] = c))).\end{aligned}$$

We define the macros as shown in Table 4.5. A node i is *conflicted* if it has a conflicting colour with at least one of its neighbours. i is *subtractable* if there is a colour value less than $i[colour]$ that i can change to without a conflict with any of its neighbours. i is *impedensable* if i is not conflicted, and it is the highest ID node that is subtractable.

| |
|--|
| $\text{CONFLICTED-GC-ELL}(i) \equiv \exists j \in Adj_i : j[colour] = i[colour].$ |
| $\text{SUBTRACTABLE-GC-ELL}(i) \equiv \exists c \in [1 : i[colour] - 1] : \forall j \in Adj_i : j[colour] \neq c.$ |
| $\begin{aligned}\text{IMPEDENSABLE-GC-ELL}(i) &\equiv \neg \text{CONFLICTED-GC-ELL}(i) \wedge \text{SUBTRACTABLE-GC-ELL}(i) \wedge \\ &(\forall j \in V(G) : \neg \text{CONFLICTED-GC-ELL}(j) \wedge (j[id] < i[id] \vee \neg \text{SUBTRACTABLE-GC-ELL}(j))).\end{aligned}$ |

Table 4.5: Macros used in the algorithm for GC.

Unlike SDMDS, MVC and MIS, in graph colouring (GC), each node is associated with a variable *colour* that can take several possible values (the domain can be as large as the set of natural numbers). As mentioned above, the action of setting a colour value to $i[colour] + i[id]$ is done whenever a conflict is detected. Effectively, this is like setting the colour to an error value such that the error value of every node is distinct in order to avoid a conflict. This error value will be reduced when node i becomes impedensable and decreases its colour.

The actions of the algorithm are shown in Algorithm 4.4. If a node i is impedensable, then it changes its colour to the minimum possible colour value. If i is conflicted, then it changes its colour value to $i[colour] + i[id]$.

Algorithm 4.4. *Rules for node i .*

| |
|--|
| $\text{CONFLICTED-GC-ELL}(i) \longrightarrow i[\text{colour}] = i[\text{colour}] + i[\text{id}].$ $\text{IMPEDENSABLE-GC-ELL}(i) \longrightarrow$ $i[\text{colour}] = \min_c \{c \in [1 : i[\text{colour}] - 1] : (\forall j \in \text{Adj}_i : j[\text{colour}] \neq c)\}.$ |
|--|

Algorithm 4.4 is an ELLSS algorithm: according to Definition 4.5, F_1 corresponds to the first action of Algorithm 4.2, F_2 corresponds to its second action, and S_f is the set of the states for which \mathcal{P}'_c holds true. Thus, starting from any arbitrary state, the algorithm eventually reaches a state where no two adjacent nodes have the same colour and no node can reduce its colour.

Lemma 4.7. *Algorithm 4.4 is a silent eventually lattice-linear self-stabilizing algorithm for graph colouring.*

Proof. In an arbitrary non-feasible state (where the input graph G does not manifest a valid colouring), there is at least one node that is conflicted. A conflicted node immediately executes the first instruction of Algorithm 4.4 and makes its colour equal to its ID plus its colour value. Since the value $i[\text{colour}] + i[\text{id}]$ by which a node updates its colour value will resolve such conflict with one adjacent node in 1 move, i will become non-conflicted in almost $\text{deg}(i)$ moves.

If the input graph G is in a feasible, but not optimal, state (where G manifests a valid colouring but some nodes can reduce their colour), then there is at least one subtractable node. This implies that there is an impedensable node i in that state (the subtractable node with the highest ID). Under Algorithm 4.4, any node will not execute until i changes its state. i is subtractable because there is a colour value c less than $i[\text{colour}]$ such that no node in Adj_i has that colour value. Thus i must execute to become non-subtractable. This shows that the second rule in Algorithm 4.4 is lattice-linear.

Since in a non-minimal, but valid, colouring, there is at least one node i that is impedensable, we have that a node will become non-subtractable in at most $\text{deg}(i)$ moves. Notice that when an impedensable node i changes its state, no node changes its state simultaneously. Also, the reduced colour will not have a conflict with any other node. Thus, no conflicts arise. Therefore, we have that Algorithm 4.4 is self-stabilizing.

When G manifests a valid non-subtractable colouring, no node is removable or addable. This shows that Algorithm 4.4 is silent. □

In Algorithm 4.4, the definition of IMPEDENSABLE relies only on the information about distance-2 neighbours. Hence, the evaluation of guards take $O(n)$ time. This algorithm converges in $2m + (n + 2m) = n + 4m$ moves.

4.6 Applying ELLSS in 2-Dominating Set Problem

The 2-dominating set (2DS) problem provides a stronger form of dominating set (DS), as compared to the usual MDS problem. In the *2-dominating set* problem, the input is a graph G with nodes having domain

$\{IN, OUT\}$. The task is to compute a set \mathcal{D} where some node $i \in \mathcal{D}$ iff $i[st] = IN$; \mathcal{D} must be computed such that there are no two nodes $j, k \in V(G)$ that are in \mathcal{D} , and a node $i \in V(G)$ that is not in \mathcal{D} , such that $\mathcal{D} \cup \{i\} \setminus \{j, k\}$ is a valid DS.

Unlike the SDMDS, MVC, MIS or GC problems that simply study the condition of their immediate neighbours before they change their state, and after they would change their state, the 2-DS problem looks one step further. Specifically, the usual MDS or MVC problems investigate the computation of any minimal DS or VC respectively, whereas the 2DS problem requires the computation of such a DS where it must not be the case that another valid DS can be computed while removing two nodes from it and adding one node to it.

The propositions \mathcal{P}'_d defines a DS, \mathcal{P}_d defines an MDS and \mathcal{P}_{2d} defines an optimal state, obtaining a 2DS. These propositions are defined below.

$$\begin{aligned}\mathcal{P}'_d(\mathcal{D}) &\equiv \forall i \in V(G) : i \in \mathcal{D} \vee (\exists j \in Adj_i : j \in \mathcal{D}). \\ \mathcal{P}_d(\mathcal{D}) &\equiv \mathcal{P}'_d(\mathcal{D}) \wedge (\forall (i \in V(G) : \neg \mathcal{P}_d(D \setminus \{i\}))). \\ \mathcal{P}_{2d}(\mathcal{D}) &\equiv \mathcal{P}_d(\mathcal{D}) \wedge \neg(\exists i \in V(G), i \notin \mathcal{D} : \\ &(\exists j, k \in Adj_i, j \in \mathcal{D}, k \in \mathcal{D} : \mathcal{P}'_d(\mathcal{D} \cup \{i\} \setminus \{j, k\})))\end{aligned}$$

Our algorithm is based on the following intuition: Let \mathcal{D} be an MDS. If there exists nodes i, j and k such that $j, k \in \mathcal{D}$ and $i \notin \mathcal{D}$, and $\mathcal{D} \cup \{i\} - \{j, k\}$ is also a DS, then j and k must be neighbours of i .

The macros that we utilize are in Table 4.6. A node i is *addable* if $i[st] = OUT$ and all the neighbours of i are also out of the DS. i is *removable* if $i[st] = IN$ and there exists at least one neighbour of i that is also in the DS. A node i is *2-addable* if $i[st] = OUT$ there exist nodes j and k in the distance-2 neighbourhood of i where $j[st] = IN$ and $k[st] = IN$ such that j and k can be removed and i can be added to the DS such that j, k and their neighbours stay dominated. A node is *unsatisfied* if it is removable or 2-addable. A node is *impedensable* if it is the highest id node in its distance-4 neighbourhood that is unsatisfied.

The algorithm for the 2-dominating set problem is as follows. If a node i is addable, then it turns itself in the DS, ensuring that i and all its neighbouring nodes stay dominated. As stated above, a node is impedensable then it is either removable or 2-addable. If a node is impedensable and removable, then it turns itself out of the DS, ensuring that i is not such a node that is not needed in the DS, but is still present in the DS. If i is impedensable and 2-addable, then there are two nodes j and k in the DS such that j and k can be removed, and i can be added, and the resulting DS is still a valid DS. In this case, i moves into the DS, and moves j and k out of the DS.

| |
|---|
| $\text{ADDABLE-2DS-ELL} \equiv i[st] = \text{OUT} \wedge (\forall j \in \text{Adj}_i : j[st] = \text{OUT}).$ |
| $\begin{aligned} \text{REMOVABLE-2DS-ELL}(i) \equiv & i[st] = \text{IN} \wedge (\forall j \in \text{Adj}_i \cup \{i\} : ((j \neq i \wedge j[st] = \text{IN}) \\ & \vee (\exists k \in \text{Adj}_j, k \neq i : k[st] = \text{IN}))). \end{aligned}$ |
| $\begin{aligned} \text{TWO-ADDABLE-2DS-ELL}(i) \equiv & i[st] = \text{OUT} \wedge (\forall j \in \text{Adj}_i^2 \cup \{i\} : \\ & \neg(\text{ADDABLE-2DS-ELL}(j) \vee \text{REMOVABLE-2DS-ELL}(j))) \wedge \\ & (\exists j, k \in \text{Adj}_i^2, j[st] = \text{IN}, k[st] = \text{IN} : \\ & (\forall q \in \text{Adj}_j \cup \text{Adj}_k \cup \{j, k\} : (\exists r \in \text{Adj}_q : r[st] = \text{IN} \vee r = i))). \end{aligned}$ |
| $\text{UNSATISFIED-2DS-ELL}(i) \equiv \text{REMOVABLE-2DS-ELL}(i) \vee \text{TWO-ADDABLE-2DS-ELL}(i).$ |
| $\begin{aligned} \text{IMPEDENSABLE-2DS-ELL}(i) \equiv & \text{UNSATISFIED-2DS-ELL}(i) \wedge (\forall j \in \text{Adj}_i^4 : \\ & (\neg \text{UNSATISFIED-2DS-ELL}(j) \vee i[id] > j[id])). \end{aligned}$ |

Table 4.6: Macros used in the algorithm for 2DS.

Algorithm 4.5. *Rules for node i .*

| |
|--|
| $\text{ADDABLE-2DS-ELL}(i) \longrightarrow i[st] = \text{IN}.$ |
| $\text{IMPEDENSABLE-2DS-ELL}(i) \longrightarrow$ |
| $\left\{ \begin{array}{ll} i[st] = \text{OUT}. & \text{if } i[st] = \text{IN}. \\ j[st] = \text{OUT}, k[st] = \text{OUT}, i[st] = \text{IN}. & \text{if } i[st] = \text{OUT}. \end{array} \right.$ |
| $// \text{The reference to } j \text{ and } k \text{ is from the definition of } \text{TWO-ADDABLE-2DS-ELL}(i)$ |

This is an ELLSS algorithm that works in three phases: first, every node i checks if it is addable. If i is not addable, then i checks if it is impedensable and removable, providing a minimal DS. And finally, i checks if it is impedensable and 2-addable, providing a 2DS. Thus, this algorithm satisfies the conditions in Definition 4.5, where F_1 constitutes of the first action of Algorithm 4.5, F_2 corresponds to its second action, and S_f is the set of the states for which \mathcal{P}'_d holds true. Thus, starting from any arbitrary state, the algorithm eventually reaches a state where \mathcal{D} is 2-dominating set.

Lemma 4.8. *Algorithm 4.2 is a silent eventually lattice-linear self-stabilizing algorithm for 2-dominating set.*

Proof. In an arbitrary non-feasible state (where the input graph G does not manifest a valid DS), there is at least one node that is addable. An addable node immediately executes the first instruction of Algorithm 4.5 and moves in the DS. This implies that by the end of the first round, we obtain a valid (possibly non-minimal) DS.

If the input graph G is in a feasible, but not optimal, state (where G manifests a non-minimal DS), then there is at least one node that is removable or 2-addable. This implies that there is at least one impedensable node i in that state (e.g., the node, which is removable or 2-addable, with the highest ID). Under Algorithm 4.2, any node in Adj_i^4 will not execute until i changes its state. If i is removable, then all its neighbours are being dominated by a node other than i . If i is 2-addable, then there exists a pair of nodes j and k such that if j and k can move out and i moves in, then all nodes in Adj_i , Adj_j and Adj_k will stay dominated, including i , j and k . Thus i must execute so that it becomes non-impedensable. This shows that the second rule in Algorithm 4.5 is lattice-linear.

Notice that if an arbitrary node j and k can move out of the DS given that all nodes stay dominated if i moves in, then j and k must be the neighbours of i . This is assuming that G is in a valid dominating set. Otherwise, it cannot be guaranteed that i can dominate the nodes that only j or k are dominating.

Since in a non-minimal, but valid, DS, there is at least one node that is removable impedensable, we have that with every move of a removable impedensable node, the size of the DS, manifested by G , reduces by 1. Now assume that G manifests a DS such that no node is addable or removable. Here, if G does not manifest a 2-dominating set, then, from the discussion from the above paragraph there must exist at least one set of three nodes i , j and k such that j and k can move out and i can move in guaranteeing that all nodes in Adj_i , Adj_j and Adj_k stay dominated, including i , j and k . With every move of a 2-addable impedensable node, the size of the DS, manifested by G , reduces by 1. Also, notice that when an impedensable node i changes its state, no node in Adj_i^4 changes its state simultaneously. Thus, the validity of the DS is not impacted when i moves. Therefore, we have that Algorithm 4.5 is self-stabilizing.

When G manifests a 2-dominating set, no node is addable, removable or 2-addable. This shows that Algorithm 4.5 is silent. □

Note that in Algorithm 4.5, the definition of REMOVABLE relies on the information about distance-2 neighbours, and consequently, the definition of TWO-ADDABLE relies on the information about distance-4 neighbours. Hence, because of the time complexity of evaluating if a node is impedensable, the guards take $O(\Delta^8)$ time. This algorithm converges in $3n$ moves (or more precisely 1 round plus $2n$ moves).

In this algorithm, one of the actions is changing the states of 3 processes at once. However, it can be implemented in a way that a process changes its own state only. We sketch how this can be done as follows. To require that a process only changes its own state, we will need additional variables so processes know that they are in the midst of an update where i needs to add itself to \mathcal{D} and j and k need to remove themselves from \mathcal{D} . Intuitively, it will need a variable of the form *getout.i* which will be set to $\{j, k\}$ to instruct j and k to leave the dominating set. When j or k are in the midst of leaving the dominating set, all the nodes in

Adj_i^6 will have to wait until the operation is completed. With this change, we note that the algorithm will not be able to tolerate incorrect initialization of $getout.i$ while preserving lattice-linearity.

4.7 Experiments

In this section, we present the experimental results of time convergence of shared memory programs. We focus on the problem of maximal independent set (Algorithm 4.3) as an example.

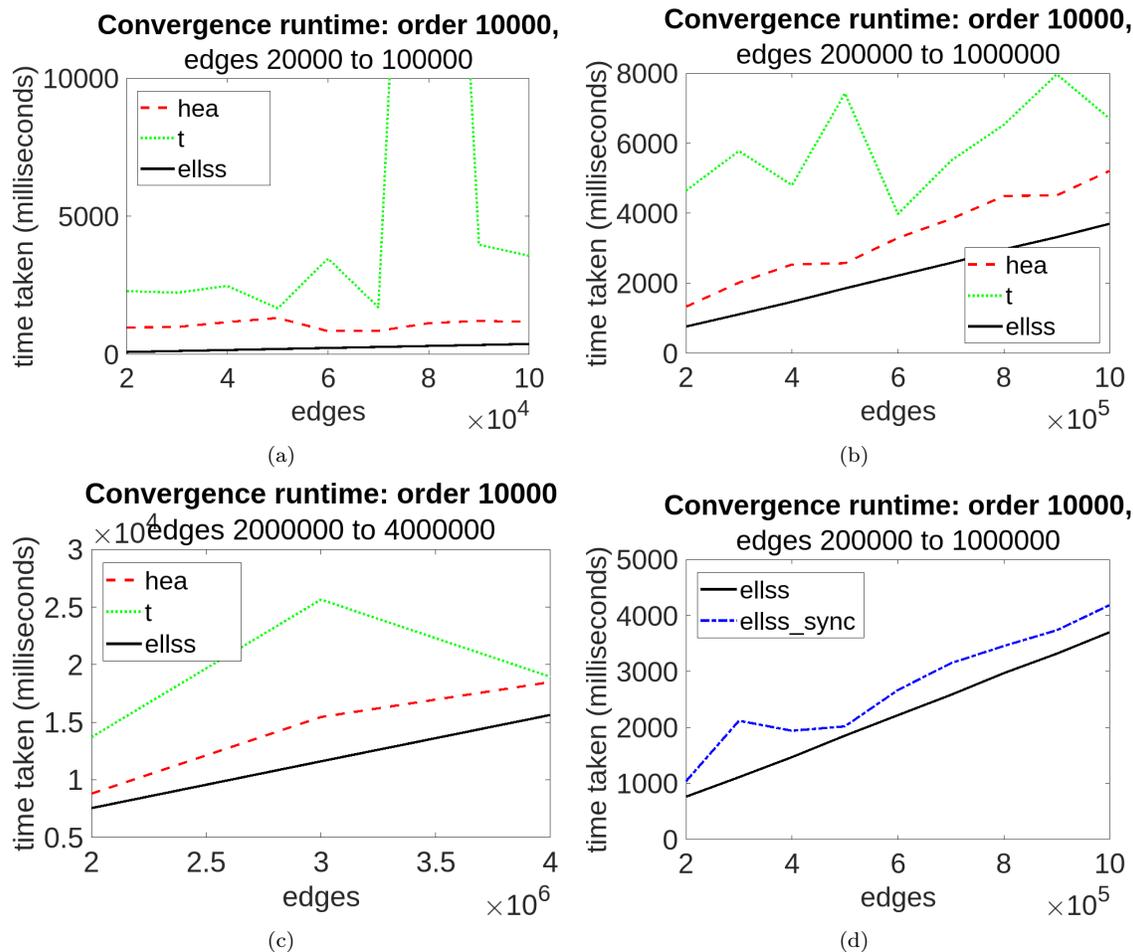


Figure 4.2: Maximal Independent set algorithms convergence time on random graphs generated by `networkx` library of `python3`. All graphs are of 10,000 nodes. Comparison between runtime of Algorithm 4.3, Hedetniemi et al. (2003) [14] (marked as *hea*) and Turau (2007) [15] (marked as *t*) and Algorithm 4.3. (a) 20,000 to 100,000 edges, Algorithm 4.3, [14] and [15]. (b) 200,000 to 1,000,000 edges, Algorithm 4.3, [14] and [15]. (c) 2,000,000 to 4,000,000 edges, Algorithm 4.3, [14] and [15]. (d) 20,000 to 100,000 edges, Algorithm 4.3 and Algorithm 4.3 lockstep synchronized.

We compare Algorithm 4.3 with the algorithms present in the literature for the maximal independent set problem. Specifically, we implemented the algorithms present in Hedetniemi et al. (2003) [14] and Turau (2007) [15], and compare their convergence time. The input graphs were random graphs of order 10,000 nodes, generated by the `networkx` library of `python`. For comparing the performance results, all algorithms

are run on the same set of graphs.

The experiments are run on Cuda using the `gcccuda2019b` compiler. The program for Algorithm 4.3 was run asynchronously, and the algorithms in [14] and [15] are run under the required synchronization model. The experiments are run on Intel(R) Xeon(R) Platinum 8260 CPU @ 2.40 GHz, cuda v1005. The programs are run using the command `nvcc <program>.cu -G`. Here, each multiprocessor ran 256 threads. And, the system provided sufficient multiprocessors so that each node in the graph can have its own thread. All the observations are an average of 16 readings.

Figure 4.2 (a) (respectively, Figure 4.2 (b) and Figure 4.2 (c)) shows a line graph comparison of the convergence time for these algorithms with the number of edges varying from 20,000 to 100,000 (respectively, 200,000 to 1,000,000 and 2,000,000 to 4,000,000). So, the average degree is varying from 4 to 20 (respectively, 40 to 200 and 400 to 800). Observe that the convergence time taken by the program for Algorithm 4.3 is significantly lower than the other two algorithms.

Next, we considered how much of the benefit of Algorithm 4.3 can be allocated to asynchrony due to the property of lattice-linearity. For this, we compared the performance of Algorithm 4.3 running in asynchrony (to allow nodes to read old/inconsistent values) and running in lock-step (to ensure that they only reads the most recent values). Figure 4.2 (d) compares these results. We observe that the asynchronous implementation has lower convergence time.

We have performed the experiments on shared memory architecture that allows nodes to access all memory *quickly*. This means that the overhead of synchronization is low. By contrast, if we had used a distributed system, where computing processors are far apart, the cost of synchronization will be even higher. Hence, the benefit of lattice-linearity (where synchronization is not needed) will be even higher.

4.8 Summary of the Chapter

We extended lattice-linearity from [6] to the context of self-stabilizing algorithms. A key benefit of lattice-linear systems is that correctness is preserved even if nodes read old information about other nodes. However, the approach in [6] relies on the assumption that the algorithm starts in specific initial states, hence, it is not directly applicable in self-stabilizing algorithms.

We began with the service demand based minimal dominating set (SDMDS) problem and designed a self-stabilizing algorithm for the same. Subsequently, we observed that it consists of two parts: One part makes sure that it gets the system to a state in S_f . The second part is a lattice-linear algorithm that constructs a minimal dominating set if it starts in some valid initial states, say a state in S_f . We showed that these parts have bounded interference, thus, they guarantee that the system stabilizes even if the nodes execute asynchronously.

We defined the general structure of eventually lattice-linear self-stabilization to capture such algorithms.

We demonstrated that it is possible to develop eventually lattice-linear self-stabilizing (ELLSS) algorithms for minimal vertex cover, maximal independent set, graph colouring and 2-dominating set problems.

We also demonstrated that these algorithms substantially benefit from their ELLSS property. They outperform existing algorithms while they guarantee convergence without synchronization among processes.

CHAPTER 5

FULLY LATTICE-LINEAR ALGORITHMS

In Chapter 4, we study algorithms that induce lattices only in a subset of the state space. The algorithms that we develop in Chapter 4 are called eventually lattice-linear algorithms. Such algorithms are capable of inducing one or more lattices in a subset of the state space, and guarantee that the system will transition from an arbitrary global state to a state in one of the lattices, and then it transitions to an optimal state traversing through that lattice.

In this chapter, we focus on studying whether lattices can be induced in the entire state space for non-lattice-linear problems. Specifically, in this chapter, we alleviate the limitations of, and bridge the gap between, [6] and Chapter 4 by introducing fully lattice-linear algorithms (FLLAs). The former creates a single lattice in the state space and does not allow self-stabilization whereas the latter creates multiple lattices in a subset of the state space. FLLAs induce one or more lattices among the reachable states and can enable self-stabilization. This overcomes the limitations of [6] and Chapter 4. We present fully lattice-linear self-stabilizing algorithms for the minimal dominating set (MDS), graph colouring (GC), minimal vertex cover (MVC) and maximal independent set (MIS) problems, and a lattice-linear 2-approximation algorithm for vertex cover (VC).

We show that the algorithm developed by Goswami et al. [16] is lattice-linear. Lattice-linearity follows that the moves of the robots are predictable. This allows us to show tighter bounds to the arena traversed by the robots under the algorithm. As a consequence of tighter bounds on this arena, (1) we obtain a better convergence time bound for this algorithm, which is lower than that showed in [16], and (2) we show that the gathering point of the robots can be uniquely determined from the initial, or any intermediate, global state. We show that this algorithm converges in $2n$ rounds, which is lower as compared to the time complexity bound ($2.5(n + 1)$ rounds) shown in [16].

The algorithms for MDS, MVC and MIS converge in n moves and the algorithm for GC converges in $n+2m$ moves. These algorithms are fully tolerant to consistency violations and asynchrony. The 2-approximation algorithm for VC is the first lattice-linear approximation algorithm for an NP-Hard problem; it converges in n moves.

The algorithms present in this chapter tolerate asynchrony in AMR model (cf. Section 2.1.1).

Organization of the Chapter

This chapter is organized as follows. In Section 5.1, we recap on eventually lattice-linear algorithms and discuss the motivation behind the theory present in this chapter. In Section 5.2, we describe the general structure of a (fully) lattice-linear algorithm. In Section 5.3, we present a fully lattice linear algorithm for

minimal dominating set. We present a fully lattice linear algorithm for graph colouring in Section 5.4.

We discuss why the design used to develop algorithms for minimal dominating set and graph colouring cannot be extended to develop algorithms for minimal vertex cover and maximal independent set in Section 5.5. We present algorithms for minimal vertex cover and maximal independent set problems in Section 5.5.2 and Section 5.5.3 respectively.

In Section 5.6, we compare the convergence speed of the algorithm presented in Section 5.3 with other algorithms (for the minimal dominating set problem) in the literature.

We present a lattice-linear 2-approximation algorithm for vertex cover in Section 5.7. In Section 5.8, we study the lattice-linearity properties of the algorithm developed by Goswami et al. [16]. Finally, we summarize the chapter in Section 5.9.

5.1 Revisiting Eventually Lattice-Linear ALGORITHMS

Unlike the lattice-linear problems where the problem description creates a lattice among the states in S , there are problems where the states do not form a lattice naturally, i.e., in those problems, given a suboptimal global state, the problem does not specify a specific set of nodes to change their state. As a result, in such problems, there are instances in which the impedensable nodes cannot be determined naturally, i.e., in those instances $\exists s : \neg \mathcal{P}(s) \wedge (\forall i : \exists s' : \mathcal{P}(s') \wedge s[i] = s'[i])$.

However, lattices can be induced in the state space algorithmically in these cases. In Chapter 4, we presented algorithms for some of such problems. Specifically, the algorithms presented in Chapter 4 partition the state space into two parts: feasible and infeasible states, and induce multiple lattices among the feasible states. These algorithms work in two phases. The first phase takes the system from an infeasible state to a feasible state (where the system starts to exhibit the desired property), which is an element of a lattice. In the second phase, only an impedensable node can change its state. This phase takes the system from a feasible state to an optimal state. These algorithms converge starting from an arbitrary state; they are called *eventually lattice-linear self-stabilizing algorithms*.

Example 5.1. MDS. *In the minimal dominating set problem, the task is to choose a minimal set of nodes \mathcal{D} in a given graph G such that for every node in $V(G)$, either it is in \mathcal{D} , or at least one of its neighbours is in \mathcal{D} . Each node i stores a variable $i[st]$ with domain $\{IN, OUT\}$; $i \in \mathcal{D}$ iff $i[st] = IN$. \square*

Remark: The minimal dominating set (MDS) problem is not a lattice-linear problem. This is because, for any given node i , an optimal state can be reached if i does or does not change its state. Thus i cannot be deemed as impedensable or not impedensable under the natural constraints of MDS.

Example 5.MDS: continuation 1. *Even though MDS is not a lattice-linear problem, lattice-linearity can be imposed on it algorithmically. Algorithm 5.1 (present in the following) is based on the algorithm in*

Chapter 4 for a more generalized version of the problem, the service demand based minimal dominating set problem. Algorithm 5.1 consists of two phases. In the first phase, if node i is addable, i.e., if i and all its neighbours are not in dominating set (DS) \mathcal{D} , then i enters \mathcal{D} . This phase does not satisfy the constraints of lattice-linearity from Definition 2.2. However, once the algorithm reaches a state where nodes in \mathcal{D} form a (possibly non-minimal) DS, phase 2 imposes lattice-linearity. Specifically, in phase 2, a node i leaves \mathcal{D} iff it is impedensable, i.e., i along with all neighbours of i stay dominated even if i moves out of \mathcal{D} , and i is of the highest ID among all the removable nodes within its distance-2 neighbourhood. \square

Algorithm 5.1. Eventually lattice-linear algorithm for MDS.

```

ADDABLE-MDS-ELL( $i$ )  $\equiv$   $i[st] = OUT \wedge (\forall j \in Adj_i : j[st] = OUT)$ .
REMOVABLE-MDS-ELL( $i$ )  $\equiv$   $i[st] = IN \wedge (\forall j \in Adj_i \cup \{i\} : ((j \neq i \wedge j[st] = IN) \vee$ 
     $(\exists k \in Adj_j, k \neq i : k[st] = IN)))$ .

// Node  $i$  can be removed without violating dominating set
IMPEDENSABLE-MDS-ELL( $i$ )  $\equiv$  REMOVABLE-MDS-ELL( $i$ )  $\wedge$ 
     $(\forall j \in Adj_i^2 : \neg$ REMOVABLE-MDS-ELL( $j$ )  $\vee i[id] > j[id])$ .

Rules for node  $i$ :
ADDABLE-MDS-ELL( $i$ )  $\longrightarrow$   $i[st] = IN$ . (phase 1)
IMPEDENSABLE-MDS-ELL( $i$ )  $\longrightarrow$   $i[st] = OUT$ . (phase 2)

```

Example 5.MDS: continuation 2. To illustrate the lattice imposed by phase 2 of Algorithm 5.1, consider an example graph G_4 with four nodes such that they form two disjoint edges, i.e., $V(G) = \{v_1, v_2, v_3, v_4\}$ and $E(G) = \{\{v_1, v_2\}, \{v_3, v_4\}\}$. Assume that G is initialized in a feasible state.

The lattices formed in this case are shown in Figure 5.1. We write a state s of this graph as $\langle v_1[st], v_2[st], v_3[st], v_4[st] \rangle$. We assume that $v_i[id] > v_j[id]$ iff $i > j$. Due to phase 1, the nodes not being dominated move in the DS, which makes the system traverse to a feasible state. Now, due to phase 2, only impedensable nodes move out, thus, lattices are induced among the feasible global states as shown in the figure. \square



Figure 5.1: The lattices induced in the problem instance in Example MDS continuation 2. Transitive edges are not shown for brevity.

In Algorithm 5.1, lattices are induced among only some of the global states. After the execution of phase 1, the algorithm locks into one of these lattices. Thereafter in phase 2, the algorithm executes lattice-linearly to reach the supremum of that lattice. Since the supremum of every lattice represents an MDS, this algorithm always converges to an optimal state.

5.2 Overcoming Limitations of [6] and Chapter 4

In this section, we introduce fully lattice-linear algorithms that induce a lattice structure among all reachable states. While defining these algorithms, we also distinguish them from the closely related work in [6] and Chapter 4. Specifically, we discuss why developing algorithms for non-lattice-linear problems (such that the algorithms are lattice-linear, i.e., they induce lattices in the reachable state space) requires the innovation presented in this chapter.

In [6], authors consider lattice-linear problems. Here, the state space is induced under a predicate and forms one lattice. Such problems, for a given suboptimal state, specify a set of nodes that must change their state, in order for the system to reach an optimal state. Such problems possess only one optimal state, and hence a violating node must change its state. The acting algorithm simply follows that lattice to reach the optimal state.

Certain problems, e.g., dominating set are not lattice-linear (cf. the remark below Example 5.1) and thus they cannot be modelled under the constraints of [6]. That is, the problem cannot specify for an arbitrary suboptimal state, a specific set of nodes that must change their state. Such problems are studied in Chapter 4. An interesting observation on the algorithms studied in Chapter 4 is that they induce multiple lattices in a subset of the state space (c.f. Figure 5.1).

Limitations of [6]

From the above discussion, we note that the general approach presented in Chapter 4 is applicable to a wider class of problems. Additionally, many lattice-linear problems do not allow self-stabilization. In such cases, e.g., in SMP, if the algorithm starts in, e.g., the supremum of the lattice, then it may terminate declaring that no solution is available. Unless the supremum is the optimal state, the acting algorithm cannot be self-stabilizing.

Limitations of Chapter 4

In eventually lattice-linear algorithms (e.g., Algorithm 5.1 for MDS), the lattice structure is imposed only on a subset of states. Thus, by design, the algorithm has a set of rules, say \mathcal{A}_1 , that operate in the part of the state space where the lattice structure does not exist, and another set of rules, say \mathcal{A}_2 , that operate in the part of the state space where the lattice is induced. Since actions of \mathcal{A}_1 operate outside the lattice structure, a developer must guarantee that if the system is initialized outside the lattice structure, then \mathcal{A}_1 converges the system to one of the states participating in the lattice (from where \mathcal{A}_2 will be responsible for

the traversal of the system through the lattice) and thus the developer faces an extra proof obligation. In addition, it also must be proven that actions of \mathcal{A}_2 and the actions of \mathcal{A}_1 do not interfere with each other. E.g., the developer (in the context of Algorithm 5.1) has to make sure that actions of \mathcal{A}_2 do not perturb the node to a state where the selected nodes do not form a dominating set.

Alleviating the Limitations of [6] and Chapter 4

In this chapter, we investigate if we can benefit from the advantages of both [6] and Chapter 4. We study if there exist fully lattice-linear algorithms where lattices can be imposed on all reachable states, forming single or multiple lattices. In the case that there are multiple optimal states and the problem requires self-stabilization, it would be necessary that multiple disjoint lattices are formed where the supremum of each lattice is an optimal state. Self-stabilization also requires that these lattices are exhaustive, i.e., they collectively contain all states in the state space.

Incorporating the property of self-stabilization ensures that the system can be allowed to initialize in any state, and asynchrony can be permitted. The initial state locks into one of the lattices, and due to the induction of \prec -lattices, such algorithms ensure a deterministic output (all local states visited by individual nodes form a total order, so an impedensable node has only one choice of action, and thus, the global state of convergence can be predicted deterministically from the initial state or any intermediate state; the following sections contain examples of such algorithms). Such algorithms would also permit multiple optimal states. In addition, there will be no need to deal with interference between actions.

Definition 5.1. Lattice-linear algorithms (LLA). *Algorithm A is an LLA for a problem P , iff there exists a predicate \mathcal{P} and A induces a \prec -lattice among the states of $S_1, \dots, S_w \subseteq S (w \geq 1)$, such that*

- *State space S of P contains mutually disjoint lattices, i.e.*
 - $S_1, S_2, \dots, S_w \subseteq S$ are pairwise disjoint.
 - $S_1 \cup \dots \cup S_w$ contains all the reachable states (starting from a set of initial states, if specified; if an arbitrary state can be an initial state, then $S_1 \cup \dots \cup S_w = S$).
- *Lattice-linearity is satisfied in each subset under \mathcal{P} , i.e.,*
 - P is deemed solved iff the system reaches a state where \mathcal{P} is true
 - $\forall k : 1 \leq k \leq w, \mathcal{P}$ is lattice-linear with respect to the \prec -lattice induced in S_k by A , i.e., $\forall s \in S_k :$
 $\neg \mathcal{P}(s) \Rightarrow \exists i : \text{IMPEDENSABLE}(i, s, \mathcal{P})$.

Remark: Any algorithm that traverses a \prec -lattice of global states is a lattice-linear algorithm. An algorithm that solves a lattice-linear problem, under the constraints of lattice-linearity, e.g. the algorithm described in Example 2.3, is also a lattice-linear algorithm.

Definition 5.2. Self-stabilizing LLA. *Continuing from Definition 5.1, A is self-stabilizing only if $S_1 \cup S_2 \cup$*

$\dots \cup S_w = S$ and $\forall k : 1 \leq k \leq w$, the supremum of the lattice induced among the states in S_k is optimal.

5.3 Fully Lattice-Linear Algorithm for Minimal Dominating Set (MDS)

In this section, we present a lattice-linear self-stabilizing algorithm for MDS. MDS has been defined in Example 5.1.

We describe the algorithm as Algorithm 5.2. The first two macros are the same as Example 5.1. The definition of a node being impedensable is changed to make the algorithm fully lattice-linear. Specifically, even allowing a node to enter into the dominating set (DS) is restricted such that only the nodes with the highest ID in their distance-2 neighbourhood can enter the DS. Any node i which is addable or removable will toggle its state iff it is impedensable, i.e., iff any other node $j \in Adj_i^2 : j[id] > i[id]$ is neither addable nor removable. In the case that i is impedensable, if i is addable, then we call it *addable-impedensable*, otherwise, if it is removable, then we call it *removable-impedensable*.

Algorithm 5.2. *Algorithm for MDS.*

$$\begin{aligned} \text{REMOVABLE-MDS-FLL}(i) &\equiv i[st] = IN \wedge (\forall j \in Adj_i \cup \{i\} : ((j \neq i \wedge j[st] = IN) \vee \\ &\quad (\exists k \in Adj_j, k \neq i : k[st] = IN))). \\ \text{ADDABLE-MDS-FLL}(i) &\equiv i[st] = OUT \wedge (\forall j \in Adj_i : j[st] = OUT). \\ \text{UNSATISFIED-MDS-FLL}(i) &\equiv \text{REMOVABLE-MDS-FLL}(i) \vee \text{ADDABLE-MDS-FLL}(i). \\ \text{IMPEDENSABLE-MDS-FLL}(i) &\equiv \text{UNSATISFIED-MDS-FLL}(i) \wedge \\ &\quad (\forall j \in Adj_i^2 : \neg \text{UNSATISFIED-MDS-FLL}(j) \vee i[id] > j[id]). \\ \text{Rules for node } i. \\ \text{IMPEDENSABLE-MDS-FLL}(i) &\longrightarrow i[st] = \neg i[st]. \end{aligned}$$

Lemma 5.1. *Any node in an input graph does not revisit its older state while executing under Algorithm 5.2.*

Proof. Let s be the global state at time t while Algorithm 5.2 is executing. We have from Algorithm 5.2 that if a node i is addable-impedensable or removable-impedensable, then no other node in Adj_i^2 changes its state.

If i is addable-impedensable at t , then any node in Adj_i is out of the DS. After when i moves in, then any other node in Adj_i is no longer addable, so they do not move in after t . As a result, i does not have to move out after moving in.

If otherwise i is removable-impedensable at t , then all the nodes in $Adj_i \cup \{i\}$ are being dominated by some node other than i . So after when i moves out, then none of the nodes in Adj_i , including i , becomes unsatisfied.

Let that i is dominated and out, and some $j \in Adj_i$ is removable impedensable. j will change its state to *OUT* only if i is being covered by another node. Also, while j turns out of the DS, no other node in Adj_j^2 , and consequently in Adj_i , changes its state. As a result, i does not have to turn itself in because of the action of j .

From the above cases, we have that i does not change its state to $i[st]$ after changing its state from $i[st]$ to $i[st']$. throughout the execution of Algorithm 5.2. \square

To demonstrate that Algorithm 5.2 is lattice-linear, we define state value and rank, the auxiliary variables associated with nodes and global states, as follows:

$$\text{STATE-VALUE-MDS}(i, s) = \begin{cases} 1 & \text{if UNSATISFIED-MDS}(i) \text{ in state } s \\ 0 & \text{otherwise} \end{cases}$$

$$\text{RANK-MDS}(s) = \sum_{i \in V(G)} \text{STATE-VALUE-MDS}(i, s).$$

Theorem 5.1. *Algorithm 5.2 is a silent self-stabilizing and lattice-linear algorithm executed by n nodes running asynchronously.*

Proof. We have from the proof of Lemma 5.1 that if G is in state s and $\text{RANK-MDS}(s)$ is non-zero, then at least one node will be impedensable, e.g., the unsatisfied node in $V(G)$ with the highest ID. For any node i , we have that $\text{STATE-VALUE-MDS}(i)$ decreases whenever i is impedensable and never increases. As a result, RANK-MDS monotonously decreases throughout the execution of the algorithm until it becomes zero. This shows that Algorithm 5.2 is self-stabilizing. Once RANK-MDS is zero, no node is impedensable, so no node makes a move. This shows that Algorithm 5.2 is silent.

Next, we show that Algorithm 5.2 is fully lattice-linear. We claim that there is one lattice corresponding to each optimal state. It follows that if there are w optimal states for a given instance, then there are w disjoint lattices S_1, S_2, \dots, S_w formed in the state space S . We show this as follows.

We observe that an optimal state (manifesting a minimal dominating set) is at the supremum of its respective lattice, as there are no outgoing transitions from an optimal state.

Furthermore, given a state s , we can uniquely determine the optimal state that would be reached from s . This is because in any given non-optimal state, the impedensable nodes, that must change their state in order to reduce the ranks of the global state of the system, can be uniquely identified. Additionally, the value, that these impedensable nodes will update their local state to, is also unique. Thus, the optimal state reached from a given state s can be uniquely identified.

This implies that starting from a state s in $S_k (1 \leq k \leq w)$, the algorithm cannot converge to any state other than the supremum of S_k . Thus, the state space of the problem is partitioned into S_1, S_2, \dots, S_w where each subset S_k contains one optimal state, say $s_{k_{opt}}$, and from all states in S_k , the algorithm converges to $s_{k_{opt}}$.

Each subset, S_1, S_2, \dots, S_w , forms a \prec -lattice where $s[i] \prec s'[i]$ if and only if $\text{STATE-VALUE-MDS}(i, s) > \text{STATE-VALUE-MDS}(i, s')$ and $s \prec s'$ iff $\text{RANK-MDS}(s) > \text{RANK-MDS}(s')$. This shows that Algorithm 5.2 is lattice-linear. \square

Example 5.MDS: continuation 3. For G_4 the lattices induced under Algorithm 5.2 are shown in Figure 5.2; each vector represents a global state $\langle v_1[st], v_2[st], v_3[st], v_4[st] \rangle$. \square

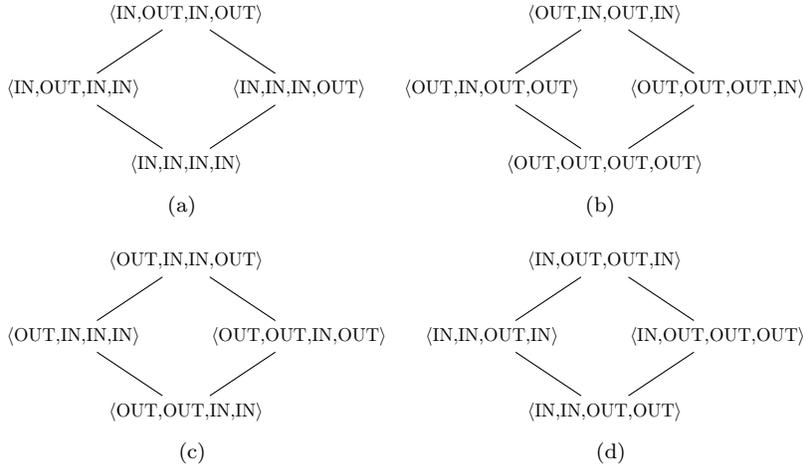


Figure 5.2: The lattices induced by Algorithm 5.2 on the graph G_4 described in Example 5.1. Transitive edges are not shown for brevity.

5.4 Fully Lattice-Linear Algorithm for Graph Colouring (GC)

In this section, we describe a fully lattice-linear algorithm for GC. We first define the GC problem, and then we describe an algorithm for GC. Graph colouring is defined in Definition 4.8 (Section 4.5).

We describe the algorithm as Algorithm 5.3. Any node i , which has a conflicting colour with any of its neighbours, or if its colour value is reducible, is an *unsatisfied* node. A node having a conflicting or reducible colour changes its colour to the lowest non-conflicting value iff it is impedensable, i.e., any node j in Adj_i with ID more than i is not unsatisfied. In the case that i is impedensable, if i has a conflict with any of its neighbours, then we call it *conflict-impedensable*, if, otherwise, its colour is reducible, then we call it *reducible-impedensable*.

Algorithm 5.3. *Algorithm for GC.*

$$\begin{aligned} \text{CONFLICTED-GC-FLL}(i) &\equiv \exists j \in \text{Adj}_i : j[\text{colour}] = i[\text{colour}]. \\ \text{REDUCIBLE-GC-FLL}(i) &\equiv \exists c \in \mathbb{N}, c < i[\text{colour}] : (\forall j \in \text{Adj}_i : c \neq j[\text{colour}]). \\ \text{UNSATISFIED-GC-FLL}(i) &\equiv \text{CONFLICTED-GC-FLL}(i) \vee \text{REDUCIBLE-GC-FLL}(i). \\ \text{IMPEDENSABLE-GC-FLL}(i) &\equiv \text{UNSATISFIED-GC-FLL}(i) \wedge \\ &\quad (\forall j \in \text{Adj}_i : \neg \text{UNSATISFIED-GC-FLL}(j) \vee i[\text{id}] > j[\text{id}]). \end{aligned}$$

Rules for node i .

$$\text{IMPEDENSABLE-GC-FLL}(i) \longrightarrow i[\text{colour}] = \min\{c \in \mathbb{N} : \forall j \in \text{Adj}_i, c \neq j[\text{colour}]\}.$$

Lemma 5.2. *Under Algorithm 5.3, the colour value may increase or decrease at its first move, after which, its colour value monotonously decreases.*

Proof. When some node i is deemed impedensable for the first time, it may be conflicted or reducible. In either case, it obtains a colour value that is not conflicting with the colour value of its neighbours. The updated colour of i will be a value from 1 to $|\text{Adj}_i| + 1$. At this time, no neighbour of i can change its colour.

Now, we show that i will not become conflicted again, after becoming impedensable for the first time. Under Algorithm 5.3, any node j in Adj_i will not change its colour until it obtains the updated colour value of i . (If j reads old information about $i[\text{colour}]$ then it will continue to wait for i to execute as required by the guard IMPEDENSABLE-GC-II.) If in case some node j in Adj_i becomes impedensable, then it must obtain a colour value that is not equal to the copy of $i[\text{colour}]$ that it reads/stores. Thus i does not become conflicted by the action of j .

Thus, after i becomes impedensable for the first time, it only reduces its colour in every subsequent move. □

To demonstrate that Algorithm 5.3 is lattice-linear, we define the state value and rank, auxiliary variables associated with nodes and global states, as follows.

$$\text{STATE-VALUE-GC}(i, s) = \begin{cases} \text{deg}(i) + 2 & \text{if } \text{CONFLICTED-GC-FLL}(i) \text{ in state } s \\ i[\text{colour}] & \text{otherwise} \end{cases}$$

$$\text{RANK-GC}(s) = \sum_{i \in V(G)} \text{STATE-VALUE-GC}(i, s).$$

Theorem 5.2. *Algorithm 5.3 is a silent self-stabilizing and lattice-linear algorithm executed by n nodes running asynchronously.*

Proof. From the proof of Lemma 5.2, we have that for any node i , STATE-VALUE-GC(i) decreases when i is impedensable and never increases. This is because i can increase its colour only once, after which it obtains a colour that is not in conflict with any of its neighbours, so any move that i makes after that will reduce its colour. Therefore, RANK-GC monotonously decreases until no node is impedensable. This shows that Algorithm 5.3 is self-stabilizing.

In any suboptimal global state, at least one node is impedensable, e.g., the highest ID node that is unsatisfied. Thus, a suboptimal global state will transition to a global state with a lesser rank. Since there are only a bounded number of colour values from 1 to $\text{deg}(i)+1$, a node can become reducible only a bounded number of times. If no node is conflicted and no node is reducible, and no node is impedensable, no node makes a move. This shows that Algorithm 5.3 is silent.

Algorithm 5.3 exhibits properties similar to Algorithm 5.2 which are elaborated in the proof for its lattice-linearity in Theorem 5.1. Thus, Algorithm 5.3 is also lattice-linear. \square

5.5 Limitations of using Simple Actions and Tiebreakers for Developing FLLA

We studied that lattice-linear algorithms for MDS and GC can be designed by simply using tie-breakers. Hence, a natural question arises if a lattice-linear algorithm can be designed for other graph theoretic problems by using some tie-breaker. The answer is no. Specifically, we cannot extend this design to develop algorithms for all graph theoretic problems – we study minimal vertex cover (MVC) and maximal independent set (MIS) problems in this context.

We first show (Section 5.5.1) the issues involved in an algorithm that simply uses a tie-breaker to decide which node enters or leaves the vertex cover. Specifically, we show that this design results in cyclic behaviour. Such behaviour is observed when we use *simple actions*, where a node only changes the state of itself when it evaluates that its guards are true, with arbitrary-distance tie-breaker. Similar results can be derived for the MIS problem. Subsequently, in Section 5.5.2 (respectively, in Section 5.5.3), we show that a lattice-linear algorithm can be developed for MVC (respectively, MIS) with *complex actions*, where a node is allowed to make changes to the variables of other nodes. Then, in Section 5.5.4, we elaborate on the properties of algorithms, that we present, for MVC and MIS.

5.5.1 Issues in Using Only a Tie-Breaker in Algorithm for Minimal Vertex Cover (MVC)

Minimal vertex cover is defined in Definition 4.6 (Section 4.3).

We could use the macros *addable* (some edge of a subject node is not covered) and *removable* (removing the node preserves the vertex cover) to design an algorithm for MVC. However, this design results in a cyclic behaviour, with respect to the local state transition of a node, even with a tie-breaker with all other nodes in the graph. To illustrate this, consider the execution of an algorithm with such a tie-breaker on a line

graph of 4 nodes (ID'd 1-4, sequentially) where all nodes are initialized to *OUT*. Here, node 4 can change its state to *IN*. Other nodes cannot change their state because there is a node with a higher ID that can enter the vertex cover. After node 4 enters the vertex cover, node 3 enters the vertex cover, as edge $\{2, 3\}$ is not covered. However, this requires node 4 to leave the vertex cover to keep it minimal.

Observe, above, that node 4 was initialized such that $4[st] = OUT$, then it changed to $4[st] = IN$ and subsequently changed again to $4[st] = OUT$. Thus, we see a cyclic behaviour which is not desired in a lattice-linear algorithm. This analysis also shows that the use of simple actions results in the system exhibiting a cyclic behaviour. However, we have that complex actions can be utilized to move around this issue, which we study in the following.

5.5.2 Fully Lattice-Linear Algorithm for Minimal Vertex Cover (MVC)

In Section 5.5.1, we discussed the issues that arise in using (1) only a tie-breaker, and (2) simple actions. Based on these limitations, in this section, we describe a lattice-linear algorithm that utilizes complex actions to solve the MVC problem.

We use the macros listed in Table 5.1. A node i is *removable* iff i is in the vertex cover, and all the neighbours of i are also in the vertex cover. i is *addable* iff i is out of the vertex cover and there is some edge $\{i, j\}$ incident on i such that j is not in the vertex cover. i is *unsatisfied* iff i is removable or i is addable. i is *impedensable* iff i is unsatisfied and there is no node j in distance-3 of i , with ID greater than i , such that j is unsatisfied. The algorithm is defined as follows. An *addable-impedensable* node i turns itself in and forces

| |
|---|
| $\text{REMOVABLE-MVC-FLL}(i) \equiv i[st] = IN \wedge (\forall j \in Adj_i : j[st] = IN).$ |
| $\text{ADDABLE-MVC-FLL}(i) \equiv i[st] = OUT \wedge (\exists j \in Adj_i : j[st] = OUT).$ |
| $\text{UNSATISFIED-MVC-FLL}(i) \equiv \text{REMOVABLE-MVC-FLL}(i) \vee \text{ADDABLE-MVC-FLL}(i).$ |
| $\text{IMPEDENSABLE-MVC-FLL}(i) \equiv \text{UNSATISFIED-MVC-FLL}(i) \wedge$ $(\forall j \in Adj_i^3 : \neg \text{UNSATISFIED-MVC-FLL}(j) \vee i[id] > j[id]).$ |

Table 5.1: Macros used in the algorithm for MVC.

all its removable neighbours out. (after accounting for the fact that i has already turned in). In this version of the algorithm, this complex action is assumed to be atomic. A *removable-impedensable* node i will move out of the vertex cover.

Algorithm 5.4. *Rules for node i .*

| |
|---|
| $\text{IMPEDENSABLE-MVC-FLL}(i) \longrightarrow$ $\left\{ \begin{array}{ll} i[st] = IN. \forall j \in Adj_i : j[st] = OUT, \text{ if } \text{REMOVABLE-MVC-FLL}(j). & \text{if } \text{ADDABLE-MVC-FLL}(i). \\ i[st] = OUT. & \text{otherwise} \end{array} \right.$ |
|---|

Lemma 5.3. *An addable (respectively, removable) node that enters (respectively, leaves) the vertex cover does not become removable (respectively addable) in any future time.*

Proof. Let s be the state at time t while Algorithm 5.4 is executing. We have from Algorithm 5.4 that if a node i is addable-impedensable or removable-impedensable, then no other node in Adj_i^3 changes its state.

If i is removable-impedensable at t , then all the nodes in Adj_i are in the vertex cover. Any node $j \in Adj_i$ cannot move out of the vertex cover until i moves out. Hence, i does not become addable after being removed at time t .

If i is addable-impedensable at t , then some node in Adj_i is out of the vertex cover. After i moves in and forces its removable neighbours out, i is no longer addable and all nodes in Adj_i are no longer removable. Now, i can become removable only if some neighbour of i enters the vertex cover. Let j be a neighbour of i . We consider two cases (1) $j[st] = OUT$ (2) $j[st] = IN$ and j was forced out by i (3) $j[st] = IN$ and j was not forced out by i .

In the first case, as long as j never enters the vertex cover, i cannot become removable, thereby ensuring that edge $\{i, j\}$ remains covered. In the event j enters the vertex cover, i may be forced out. However, (as a result) i does not become removable-impedensable.

In the second case, j is forced out of the vertex cover. This implies that all neighbours of j are already in the vertex cover. Hence, j does not become addable again. (This argument is the same as the above argument that showed that if a node is removed from the vertex cover, it does not become addable.) Since j is never added back to the vertex cover, i cannot become removable because of the action of j .

In the third case, even if j moves out afterwards, it cannot make i removable, so after time t , nodes in Adj_i do not move in; as a result, i does not have to move out after moving in. In addition, the nodes that i turned out of the vertex cover do not have to move in after t , because the neighbours of those nodes are not changing their states simultaneously when their states are being changed.

From the above cases, we have that i does not change its state to $i[st]$ after changing its state from $i[st]$ to $i[st']$. throughout the execution of Algorithm 5.4. \square

To demonstrate that Algorithm 5.4 is lattice-linear, we define state value and rank, auxiliary variables associated with nodes and global states, as follows:

$$\text{STATE-VALUE-MVC}(i, s) = \begin{cases} 1 & \text{if UNSATISFIED-MVC-FLL}(i) \text{ in state } s \\ 0 & \text{otherwise} \end{cases}$$

$$\text{RANK-MVC}(s) = \sum_{i \in V(G)} \text{STATE-VALUE-MVC}(i, s).$$

Theorem 5.3. *Algorithm 5.4 is a silent self-stabilizing and lattice-linear algorithm executed by n nodes running asynchronously.*

Proof. We have from the proof of Lemma 5.3 that if G is in state s and $\text{RANK-MVC}(s)$ is non-zero, then at least one node will be impedensable, e.g., an unsatisfied node with the highest ID. For any node i , we have that $\text{STATE-VALUE-MVC}(i)$ decreases whenever i is impedensable and never increases. In addition, by the action of i , the state value of any other node in G does not increase. (Note that adding node i to the vertex cover may have caused a neighbour j of i to be removable. If this happens, j is removed from the vertex cover in the same action atomically. Hence, j does not become unsatisfied.) In effect, RANK-MVC monotonously decreases throughout the execution of the algorithm until it becomes zero. This shows that Algorithm 5.4 is self-stabilizing. Once RANK-MVC is zero, no node is impedensable, so no node makes a move. This shows that Algorithm 5.4 is silent.

Algorithm 5.4 exhibits properties similar to Algorithm 5.2 (as well as Algorithm 5.3) which are elaborated in the proof for its lattice-linearity in Theorem 5.1. From there, we obtain that Algorithm 5.4 also is lattice-linear. \square

Example 5.2. *Let G_4^p be a graph of four vertices forming a path $\langle v_1, v_2, v_3, v_4 \rangle$ such that $v_i[id] > v_j[id]$ iff $i > j$. In Figure 5.3, we show all possible state transitions that G_4^p can go through under Algorithm 5.4. The global states in the figure are of the form $\langle v_1[st], v_2[st], v_3[st], v_4[st] \rangle$.* \square

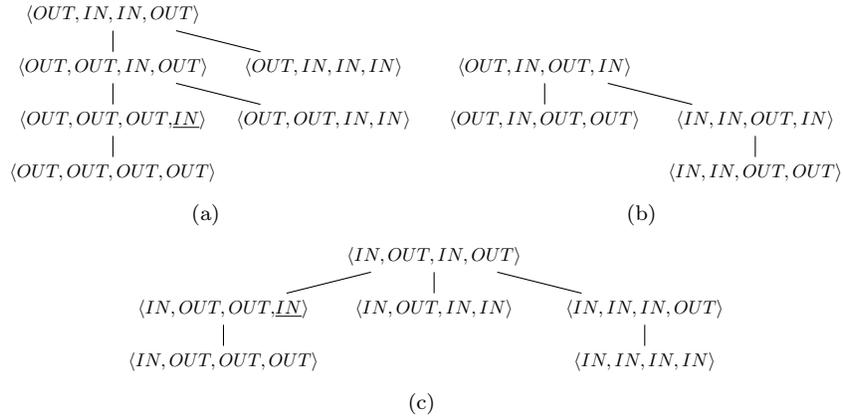


Figure 5.3: \prec -lattices formed by the global states of G_4^p of 4 nodes forming a straight path $\langle v_1, v_2, v_3, v_4 \rangle$ under Algorithm 5.4. The nodes that are kicked out, by a node that decides to move into the vertex cover, are underlined.

5.5.3 Fully Lattice-Linear Algorithm for Maximal Independent Set (MIS)

In this subsection, we describe an algorithm for the MIS. The issues, similar to the issues that we discussed in Section 5.5.1, can be observed for MIS problem as well. For example, a similar behaviour can be observed

on a path of 4 nodes, all initialized to IN . However, we can follow the general design of Algorithm 5.4, that we described in Section 5.5.2, to develop an algorithm for MIS. MIS problem is defined in Definition 4.7 (Section 4.4).

The macros that we use here are similar to the macros we used for MVC, but with opposite polarity. We use the macros listed in Table 5.2. A node i is *addable* iff i is out of the independent set (IS), and all the neighbours of i are also out of the IS. i is *removable* iff i is in the IS and there is some edge $\{i, j\}$ incident on i such that j is also in the IS. i is *unsatisfied* iff i is removable or i is addable. i is *impedensable* iff i is unsatisfied and there is no node j in distance-3 of i , with ID greater than i , such that j is unsatisfied.

| |
|---|
| $\text{ADDABLE-MIS-FLL}(i) \equiv i[st] = OUT \wedge (\forall j \in Adj_{i,j}[st] = OUT).$ |
| $\text{REMOVABLE-MIS-FLL}(i) \equiv i[st] = IN \wedge (\exists j \in Adj_i : j[st] = IN).$ |
| $\text{UNSATISFIED-MIS-FLL}(i) \equiv \text{REMOVABLE-MIS-FLL}(i) \vee \text{ADDABLE-MIS-FLL}(i).$ |
| $\text{IMPEDENSABLE-MIS-FLL}(i) \equiv \text{UNSATISFIED-MIS-FLL}(i) \wedge$ $(\forall j \in Adj_i^2 : \neg \text{UNSATISFIED-MIS-FLL}(j) \vee i[id] > j[id]).$ |

Table 5.2: Macros used in the algorithm for MIS.

Given the similarities in the MVC and MIS problems, the algorithm we develop here is similar to the algorithm we developed for MVC. The algorithm is defined as follows. A *removable-impedensable* node i turns itself out and moves all its addable neighbours into the independent set (after accounting for the fact that i has already turned out). An *addable-impedensable* node i will turn itself into the independent set. In the present design of the algorithm, the execution while i is impedensable is assumed to be executed atomically.

Algorithm 5.5. *Rules for node i .*

| |
|--|
| $\text{IMPEDENSABLE-MIS-FLL}(i) \longrightarrow$ $\begin{cases} i[st] = OUT. \forall j \in Adj_i : j[st] = IN, \text{ if } \text{ADDABLE-MIS-FLL}(j). & \text{if } \text{REMOVABLE-MIS-FLL}(i). \\ i[st] = IN. & \text{otherwise} \end{cases}$ |
|--|

Since the behaviour of Algorithm 5.5 is similar to the behaviour of Algorithm 5.4, we briefly cover the description of the behaviour of Algorithm 5.5 in the following: Lemma 5.4 and Theorem 5.4.

Lemma 5.4. *Any node in an input graph does not revisit its older state while executing under Algorithm 5.5.*

Proof. Let s be the state at time t while Algorithm 5.5 is executing. We have from Algorithm 5.5 that if a node i is addable-impedensable or removable-impedensable, then no other node in Adj_i^3 changes its state.

If i is removable-impedensable at t , then some node in Adj_i is in the independent set. After when i moves out, and turns its addable neighbours in, then we have that i is no longer removable and all nodes in Adj_i are no longer addable, so after time t , nodes in Adj_i do not move out; as a result, i does not have to move in after moving out. In addition, the nodes that i turned into the independent set do not have to move out after t , because the neighbours of those nodes are not changing their states simultaneously when their states are being changed.

Let that at some instance of time, two nodes i and j simultaneously evaluate that they are removable-impedensable. Since no nodes in Adj_i^3 (respectively, Adj_j^3) change their state under Algorithm 5.5 until i (respectively, j) changes its state, we have that i and the nodes that i would move into the independent set are not adjacent to j or the nodes that j would move into the independent set. Thus, no node by the action of i becomes removable.

If otherwise i is addable-impedensable at t , then all the nodes in Adj_i are out of IS. So after when i moves in, none of the non-unsatisfied nodes in Adj_i (non-addable nodes in Adj_i), including i , becomes unsatisfied (addable). As a result, i does not have to move out after moving in.

From the above cases, we have that i does not change its state to $i[st]$ after changing its state from $i[st]$ to $i[st']$. throughout the execution of Algorithm 5.5. \square

To demonstrate that Algorithm 5.5 is lattice-linear, we define state value and rank, auxiliary variables associated with nodes and global states, as follows:

$$\text{STATE-VALUE-MIS}(i, s) = \begin{cases} 1 & \text{if UNSATISFIED-IS}(i) \text{ in state } s \\ 0 & \text{otherwise} \end{cases}$$

$$\text{RANK-MIS}(s) = \sum_{i \in V(G)} \text{STATE-VALUE-MIS}(i, s).$$

Theorem 5.4. *Algorithm 5.5 is a silent self-stabilizing and lattice-linear algorithm executed by n nodes running asynchronously.*

Proof. We have from the proof of Lemma 5.4 that if G is in state s and $\text{RANK-MIS}(s)$ is non-zero, then at least one node will be impedensable, e.g., an unsatisfied node with the highest ID. For any node i , we have that $\text{STATE-VALUE-MIS}(i)$ decreases whenever i is impedensable and never increases. In addition, by the action of i , the state value of any other node in G does not increase. In effect, RANK-MIS monotonously decreases throughout the execution of the algorithm until it becomes zero. This shows that Algorithm 5.5 is self-stabilizing. Once RANK-MIS is zero, no node is impedensable, so no node makes a move. This shows that Algorithm 5.5 is silent.

Algorithm 5.5 exhibits properties similar to Algorithm 5.2 (as well as Algorithm 5.3 and Algorithm 5.4) which are elaborated in the proof for its lattice-linearity in Theorem 5.1. From there, we obtain that Algorithm 5.5 also is lattice-linear. \square

5.5.4 Complex Actions: Properties Shared by Algorithms for MVC and MIS

In this subsection, we study some behavioural aspects of the algorithm for MVC present in Section 5.5.2. Consequently, similar arguments for the algorithm for MIS will follow.

Algorithm 5.4 can be transformed into a simple action algorithm as follows. To accommodate that, we use the variable $i[addable]$ to set to be *true* so that the surrounding nodes can then evaluate if they are removable. We use the following additional guards. For a node i , *else-pointed* is true iff a node j in Adj_i^4 moved into the VC (and set $j[addable]$ to *true*), and there is a node k in Adj_j that is removable.

$$\text{ELSE-POINTED-MVC-FLL}(i) \equiv \exists j \in Adj_i^4 : (j[addable] = true \wedge \exists k \in Adj_j : \text{REMOVABLE-MVC-FLL}(k))$$

Consequently, the algorithm can be modified as follows. A node i will not be enabled (impedensable) if *else-pointed* is true for i . The modified algorithm that allows simple actions, thus, is defined as follows.

Algorithm 5.6. *Transformed Algorithm 5.4, where nodes only execute simple actions.*

$$\begin{aligned} \text{IMPEDENSABLE-MVC-FLL-II}(i) &\equiv \\ &(\exists j \in Adj_i : j[addable] = true \wedge \text{REMOVABLE-MVC-FLL}(i)) \vee \\ &(\neg \text{ELSE-POINTED-MVC-FLL}(i) \wedge (\text{UNSATISFIED-MVC-FLL}(i) \wedge \\ &(\forall j \in Adj_i^3 : \neg \text{UNSATISFIED-MVC-FLL}(j)) \vee \\ &i[id] > j[id])). \end{aligned}$$

Rules for node i :

$$\text{IMPEDENSABLE-MVC-FLL-II}(i) \longrightarrow \begin{cases} i[addable] = true & \text{if } i[st] = OUT. \\ i[addable] = false & \text{if } i[st] = IN. \\ i[st] = \neg i[st] & \text{unconditionally.} \end{cases}$$

Next, we identify why Algorithm 5.6 can be reconciled with the inability to design an algorithm with simple actions from Section 5.5.1. This analysis also helps us to obtain the correctness proof of Algorithm 5.6.

Reconciling Section 5.5.1 and Algorithm 5.6

As discussed in Section 5.5.1, if a tie-breaker in conjunction with simple actions is deployed, then the system would exhibit cyclic behaviour. On the other hand, Algorithm 5.6 uses only simple actions. We identify the subtlety, involved in both these results, that make this possible, in the following.

To explain how these results can coexist together, we describe the behaviour of Algorithm 5.6 with an example. Assume that this algorithm is deployed on a graph of 4 nodes forming a path, with node IDs being in the sequence $\langle 1, 4, 3, 2 \rangle$. In a simple algorithm that uses a tiebreaker with all nodes (which is considered in Section 5.5.1), node 4 would execute first then node 3, then node 2 and finally node 1. This execution order is not preserved in Algorithm 5.6, which we discuss as follows.

Specifically, let the initial global state in this graph be $\langle IN, OUT, OUT, OUT \rangle$. First, node 4 will move into the VC. Now, the node that is unsatisfied and has the highest ID is node 3. However, $ELSE\text{-}POINTED(3)$ is true, because $4[addable]$ is set to *true* and node 1 needs to move out of the vertex cover as part of the action that allowed node 4 to enter the vertex cover. In other words, node 3 can execute only after node 1 leaves the vertex cover. This is not permitted in a algorithm that uses simple actions with tie-breaker on node IDs. This effect is similar to that of priority inheritance, where node 1 inherits the priority of node 4 because it has to be forced out of the vertex cover by node 4. Hence, in this specific case, node 1 has a higher priority of movement as compared to node 3, despite the fact that node 3 is unsatisfied and is of a higher ID, because of a recent action committed by node 4.

After node 1 moves out, node 3 moves in, and thence, the system reaches an optimal state.

Correctness of Algorithm 5.6

Algorithm 5.5 is lattice-linear with respect to state value and rank, defined as follows.

$$\text{STATE-VALUE-MVC-II}(i, s) = \begin{cases} |Adj_i + 1| & \text{if UNSATISFIED-MVC-FLL}(i) \text{ in state } s \\ |\{j \in Adj_i : \text{UNSATISFIED-MVC-FLL}(j)\}| & \text{if in state } s, \neg \text{UNSATISFIED-MVC-FLL}(i) \wedge \\ & (\exists j \in Adj_i : \text{UNSATISFIED-MVC-FLL}(j)) \\ 0 & \text{otherwise} \end{cases}$$

$$\text{RANK-MVC-II}(s) = \sum_{i \in V(G)} \text{STATE-VALUE-MVC-II}(i, s).$$

Since the working of this algorithm straightforwardly follows from the working of Algorithm 5.4, we omit the proof of correctness of this algorithm. A similar algorithm can be developed for MIS, that deploys only simple actions.

5.6 Experiments

In this section, we present the experimental results of convergence times from implementations run on real-time shared memory model. We implement the algorithm for minimal dominating set (Algorithm 5.2), and compare it to algorithms by Hedetniemi et al. (2003) [14] and Turau (2007) [15]. We also present the runtime of a distance-1 transformation of Algorithm 5.2. First, we present the transformation of Algorithm 5.2 in the following subsection.

5.6.1 Transforming Algorithm 5.2 to Distance-1

In Algorithm 5.2, we observe that the guards of a node i are distance-4. First, we transform this algorithm to a distance-1 algorithm. To accomplish this, the nodes maintain additional variables, that provide them information about other nodes, as required. We use additional variables and guards to propagate this information. Due to the constraint of reading only distance-1 neighbours, the nodes may end up reading old information about the other nodes. However, due to lattice-linearity, such executions stay to be correct.

A straightforward transformation would require each node i to maintain copies of all the variables of its distance-4 neighbours. However, we use only 4 additional variables. Note that the requirement of these four variables is independent of the number of nodes in Adj_i^4 .

We use $i[l\text{dom}]$ and $i[h\text{dom}]$ to assist in propagating the information about the macro REMOVABLE-DS(i). $i[h\text{dom}]$ stores the highest ID dominator of i : it is a node in N_i of highest ID such that its state is IN . $i[l\text{dom}]$ stores the lowest ID dominator of i : it is a node in N_i of lowest ID such that its state is IN . (If such a node does not exist, these variables are set to \top (*null*).)

Once Removable-DS is transformed to a distance-1 macro, unsatisfied-ds will also be a distance-1 macro, as ADDADBLE-DS is already a distance-1 macro.

We use $i[u\text{flag}]$ and $i[h\text{ud}1]$ to assist in propagating the information about IMPEDENSABLE-II-DS. Node i sets $i[u\text{flag}]$ to *true* to indicate that i is unsatisfied. $i[h\text{ud}1]$ stores the highest ID node in distance-1, i.e., in N_i , of i that is unsatisfied.

Now, we describe the actions of the transformed distance-1 algorithm. We use the macros listed in Table 5.3. i is *hdom-outdated* iff $i[h\text{dom}]$ is not equal to the highest ID dominator of i . i is *ldom-outdated* iff $i[l\text{dom}]$ is not equal to the lowest ID dominator of i . i is *removable* iff every node $j \in N_i$ will stay dominated even if i moves out of the dominating set. This will happen if either $j[st] = IN$, or, either $j[h\text{dom}]$ or $j[l\text{dom}]$ differs from i . Node i is *addable* if all nodes in Adj_i , along with i , are out of the DS. i is *unsatisfied* if i is removable or addable. i is *unsatisfied-flag-outdated* iff $i[u\text{flag}]$ is not equivalent to i being unsatisfied. i is *hud1-outdated* iff $i[h\text{ud}1]$ is not equal to the highest ID node in N_i that is unsatisfied. i is *impedensable* if i is the highest ID node in the distance-2 neighbourhood that is unsatisfied. i is *impedensable*

iff i is hdom-outdated, ldom-outdated, unsatisfied-flag-outdated, hud1-outdated or unsatisfied-impedensable.

| |
|---|
| variables of i : $st, ldom, hdom, uflag, hud1$ |
| $\text{HDOM-OUTDATED}(i) \equiv i[hdom] \neq \arg \max\{x[id] : x \in N_i \wedge x[st] = IN\}$. |
| $\text{LDM-OUTDATED}(i) \equiv i[ldom] \neq \arg \min\{x[id] : x \in N_i \wedge x[st] = IN\}$. |
| $\text{REMOVABLE-MDS-D} \equiv i[st] = IN \wedge (\forall j \in N_i : ((j \neq i \wedge j[st] = IN) \vee ((j[ldom] \neq i \wedge j[ldom] \neq \top) \vee (j[hdom] \neq i \wedge j[hdom] \neq \top))))$. |
| $\text{ADDABLE-MDS-D}(i) \equiv i[st] = OUT \wedge (\forall j \in Adj_i : j[st] = OUT)$. |
| $\text{UNSATISFIED-MDS-D}(i) \equiv \text{REMOVABLE-MDS-D}(i) \vee \text{ADDABLE-MDS-D}(i)$. |
| $\text{UNSATISFIED-FLAG-OUTDATED}(i) \equiv i[uflag] \neq \text{UNSATISFIED-MDS-D}(i)$. |
| $\text{HUD1-OUTDATED}(i) \equiv i[hud1] \neq \arg \max\{x[id] : x \in N_i \wedge x[uflag] = true\}$. |
| $\text{UNSATISFIED-IMPEDENSABLE-MDS-D}(i) \equiv i[uflag] \wedge (\forall j \in Adj_i : (j[uflag] \Rightarrow j[id] < i[id]) \wedge (j[hud1] \neq \top \Rightarrow j[hud1] < i[hud1]))$. |
| $\text{IMPEDENSABLE-MDS-D}(i) \equiv \text{HDOM-OUTDATED}(i) \vee \text{LDM-OUTDATED}(i) \vee$ $\text{UNSATISFIED-FLAG-OUTDATED}(i) \vee \text{HUD1-OUTDATED}(i) \vee$ $\text{UNSATISFIED-IMPEDENSABLE-MDS-D}(i)$. |

Table 5.3: Macros used in transformed algorithm for MDS.

We describe the algorithm as follows. If i is hdom-outdated, then it updates $i[hdom]$ with ID of the highest ID node in N_i that is in the dominating set. If i is ldom-outdated, then it updated $i[ldom]$ with ID of the lowest ID node in N_i that is in the dominating set. If i is unsatisfied-flag-outdated, then it updates $i[uflag]$ to correctly denote whether i is unsatisfied. If i is hud1-outdated, then i updates $i[hud1]$ with the ID of the highest ID node in N_i that is unsatisfied. If i is unsatisfied-impedensable, then i toggles $i[st]$.

Algorithm 5.7. *Rules for node i .*

| |
|---|
| $\text{IMPEDENSABLE-DS-M}(i) \longrightarrow$ |
| $\left\{ \begin{array}{ll} hdom = \arg \max\{x[id] : x \in N_i \wedge x[st] = IN\} & \text{if } \text{HDOM-OUTDATED}(i). \\ ldom = \arg \min\{x[id] : x \in N_i \wedge x[st] = IN\} & \text{if } \text{LDM-OUTDATED}(i). \\ uflag \neq \text{UNSATISFIED-MDS-D}(i) & \text{if } \text{UNSATISFIED-FLAG-IMPEDENSABLE}(i). \\ i[hud1] = \arg \max\{x[id] : x \in N_i \wedge x[uflag] = true\} & \text{if } \text{HUD1-OUTDATED}(i). \\ i[st] = \neg i[st] & \text{if } \text{UNSATISFIED-IMPEDENSABLE}(i). \\ i[uflag] = false & \text{unconditionally.} \end{array} \right.$ |

Deploying the above algorithm reduces the work complexity of evaluating the guards for a node to $O(\Delta)$, which was originally $O(\Delta^4)$ in Algorithm 5.2. In principle, all lattice-linear distance- x (where $x > 1$) algorithms can be transformed into distance-1 algorithms by keeping a copy of all variables in Adj^x [17]. However, it will increase the space complexity of every node by $|Adj^x|$, without decreasing the time complexity of the evaluation of guards. By contrast, the above algorithm increases the space complexity by only $O(1)$, while decreasing the time complexity from $O(\Delta^4)$ to $O(\Delta)$.

5.6.2 Runtime Comparison

While we see a significant reduction of the time complexity, of the evaluation of guards by a node, from $O(\Delta^4)$ in Algorithm 5.2 to $O(\Delta)$ in Algorithm 5.7, it is also worthwhile to compare the convergence time of these algorithms when they are implemented on real-time systems. In this subsection, we compare the runtime of Algorithm 5.7 with Algorithm 5.2 and other algorithms.

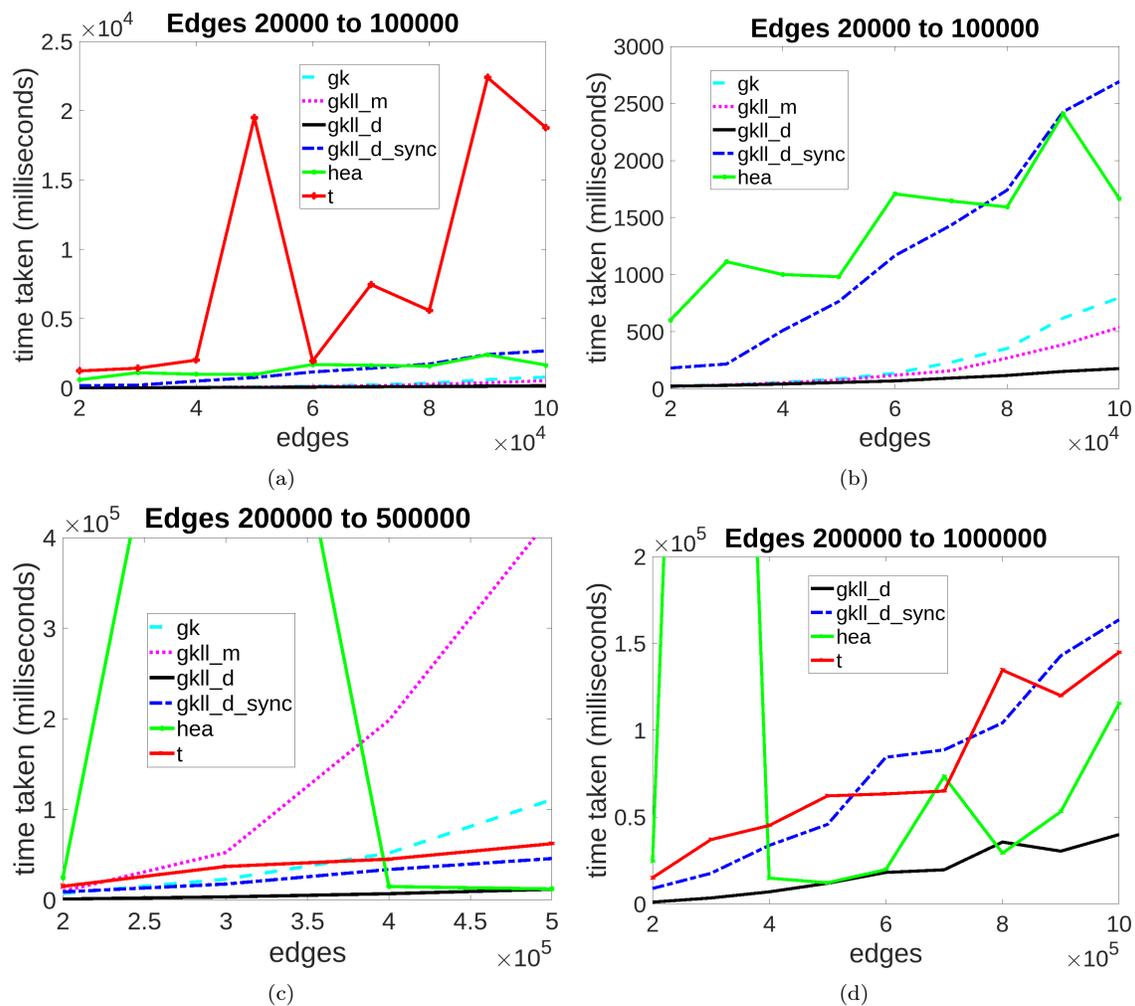


Figure 5.4: Runtime comparison of Algorithm 5.7, Algorithm 5.2, Algorithm 5.1 and other algorithms for minimal dominating set in the literature. All graphs are of 10,000 nodes.

We implemented Algorithm 5.2 (`gkll_m`), Algorithm 5.7 (`gkll_d`), lockstep synchronized Algorithm 5.7 (`gkll_d_sync`), Algorithm 5.1 (`gk`), algorithms for minimal dominating set present in Hedetniemi et al. (2003) [14] (`hea`) and Turau (2007) [15] (`t`), and compare their convergence time. The input graphs were random graphs of order 10,000 nodes, generated by the `networkx` library of python. For comparing the performance results, all algorithms are run on the same set of graphs.

The experiments are run on Cuda using the `gccuda2019b` compiler. `gkll_m`, `gkll_d` and `gk` were run asynchronously, and the algorithms in `gkll_d_sync`, `hea` and `t` were run under the required synchronization model. The experiments are run on Intel(R) Xeon(R) Platinum 8260 CPU @ 2.40 GHz, `cuda V100S`. The programs are run using the command `nvcc <program>.cu`. Here, each multiprocessor ran 256 threads. And, the system provided sufficient multiprocessors so that each node in the graph can have its own thread. All the observations are an average of 3 readings.

Figure 5.4 (a) (respectively, Figure 5.4 (c) and Figure 5.4 (d)) shows a line graph comparison of the convergence time for these algorithms with the number of edges varying from 20,000 to 100,000 (respectively, 200,000 to 500,000 and 200,000 to 1,000,000). So, the average degree is varying from 4 to 20 (respectively, 40 to 100 and 40 to 200). Figure 5.4 (b) is same as Figure 5.4 (a), except that the curve for `hea` is removed so that the other curves can be analyzed closely. Similarly, Figure 5.4 (c) and Figure 5.4 (d) are similar, however, (1) Figure 5.4 (c) shows curves for convergence time of graphs of average degree 40 to 100, whereas Figure 5.4 (d) shows curves for convergence time of graphs of average degree 40 to 200, and (2) Figure 5.4 (c) contains all 6 curves, whereas Figure 5.4 (d) does not contain curves for `gkll_m` and `gk`. Observe that the convergence time taken by the program for `gkll_d` is consistently lower than the other algorithms.

In Figure 5.4 (b), it can be observed that the runtime of `gkll_m` is lower than the other algorithms (except `gkll_d`, which is not surprising). However, in Figure 5.4 (c), it can be observed that the runtime of `gkll_m` increases more rapidly and overtakes the runtime of other algorithms (that is why we omitted `gkll_m` from Figure 5.4 (d)). This happens mainly because the nodes under `gkll_m` are reading values of nodes at distance-4 from themselves. `gk` also converges comparatively quicker than (`gkll_m`) (but not quicker than other algorithms) because its first phase is quicker: the addable nodes move in the dominating set “carelessly”, whereas in `gkll_m` the nodes moving in are “careful” as well as the nodes that move out of the dominating set, which adds to the convergence time in the case of `gkll_m`.

Next, we discuss how much of the benefit of `gkll_d` can be allocated to asynchrony due to the property of lattice-linearity. For this, observe the performance of `gkll_d` running in asynchrony (to allow nodes to read old/inconsistent values) against `gkll_d_sync` (which is the same algorithm as `gkll_d` but running in lock-step, to ensure that the nodes only read the most recent values). We observe that the asynchronous implementation has a lower convergence time. This happens mainly because both the asynchronous and the synchronized

algorithms have the same convergence time complexity, however, the cost of synchronization (time spent in synchronization, plus the requirement of at least one scheduling thread) is eliminated.

We have performed the experiments on shared memory architecture that allows the nodes to access memory *quickly*. This means that the overhead of synchronization is low. By contrast, if we had implemented these algorithms on a distributed system instead, where computing processors are placed remotely, the cost of synchronization would be even higher. Hence, we anticipate the benefit of lattice-linearity (where synchronization is not needed) to be even higher.

5.7 Lattice-Linear 2-Approximation Algorithm for Vertex Cover (VC)

It is highly alluring to develop parallel processing approximation algorithms for NP-Hard problems under the paradigm of lattice-linearity. In fact, it has been an open question if this is possible [6]. We observe that this is possible. In this section, we present a lattice-linear 2-approximation algorithm for VC.

The following algorithm is the classic 2-approximation algorithm for VC. *Choose an uncovered edge $\{A, B\}$, select both A and B , repeat until all edges are covered.* Since the minimum VC must contain either A or B , the selected VC is at most twice the size of the minimum VC.

While the above algorithm is sequential in nature, we demonstrate that we can transform it into a distributed algorithm under the paradigm of lattice-linearity as shown in Algorithm 5.8 (we note that this algorithm is *not* self-stabilizing).

In Algorithm 5.8, all nodes are initially out of the VC, and are not *done*. In the algorithm, each node will check if all edges incident on it are covered. A node is called *done* if it has already evaluated if all the edges incident on it are covered. A node is impedensable if it is not done yet and it is the highest ID node in its distance-3 neighbourhood which is not done. If an impedensable node i has an uncovered edge, and assume that $\{i, j\}$ is an uncovered edge with j being the highest ID node in Adj_i which is out (note that i is out), then i turns both i and j into the VC. If otherwise i evaluates that all its edges are covered, then it declares that it is done (i sets $i[done]$ to true), while staying out of the VC.

This is straightforward from the 2-approximation algorithm for VC. We have chosen 3-neighbourhood to evaluate impedensable to ensure that no conflicts arise while execution from the perspective of the 2-approximation algorithm for VC.

Algorithm 5.8. *2-approximation lattice-linear algorithm for VC.*

Init: $\forall i \in V(G), i[st] = OUT, i[done] = false.$

IMPEDENSABLE-VC2A(i) $\equiv i[done] = false \wedge (\forall j \in Adj_i^3 : j[id] < i[id] \vee j[done] = true).$

Rules for node $i.$

IMPEDENSABLE-VC2A(i) \longrightarrow

IF $(\forall k \in Adj_i, k[st] = IN),$ then $i[done] = true.$

ELSE, then

$j = \arg \max\{x[id] : x \in Adj_i \wedge x[done] = false\}.$

$i[st] = IN.$

$j[st] = IN, j[done] = true.$

$i[done] = true.$

Observe that the action of node i is selecting an edge $\{i, j\}$ and adding i and j to the VC. This follows straightforwardly from the classic 2-approximation algorithm.

5.7.1 Lattice-Linearity of Algorithm 5.8

To demonstrate that Algorithm 5.8 is lattice-linear, we define the state value and rank, auxiliary variables associated with nodes and global states, as follows.

$$\text{STATE-VALUE-VC2A}(i, s) = \begin{cases} |\{j \in Adj_i : s[j[st]] = OUT\}| & \text{if } s[i[st]] = OUT \\ 0 & \text{otherwise} \end{cases}$$

$$\text{RANK-VC2A}(s) = \sum_{i \in V(G)} \text{STATE-VALUE-VC2A}(i, s).$$

Theorem 5.5. *Algorithm 5.8 is a lattice-linear 2-approximation algorithm for VC.*

Proof. Lattice-linearity: In the initial state, every node i has $i[done] = false$ and $i[st] = OUT$. Let s be an arbitrary state at the beginning of some time step while the algorithm is under execution such that s does not manifest a vertex cover. Let i be some node such that i is of the highest ID in its distance 3 neighbourhood such that some of its edges are not covered. Also, let j be the node of the highest ID in Adj_i for which $j[done] = false$, if one such node exists. We have that i is the only impedensable node in its distance-3 neighbourhood, and j is the specific additional node, which i turns in. Thus the states form a partial order where state s transitions to another state s' where $s \prec s'$ and for any such $i, s'[i[st]] = IN \wedge s'[i[done]] = true$ and $s'[j[st]] = IN \wedge s'[j[done]] = true.$

If s manifests a vertex cover, then no additional nodes will be turned in, and atmost one additional node (node i , as described in the paragraph above) will have $s'[i[done]] = true$.

From the above observations, we have that for every node i , STATE-VALUE-VC2A(i) is initially $deg(i)$; this value decreases monotonously and never increases; it becomes 0 after when i is impedensable. As a result, RANK decreases monotonously until it becomes zero, because if RANK is not zero, then at least one node is impedensable (e.g., node of highest ID which has at least one uncovered edge). Thus, we have that Algorithm 5.8 also is lattice-linear. However, it induces only one lattice among the global states since the initial state is predetermined, thus $w = 1$.

2-approximability: If a node i is impedensable and if one of its edges is uncovered, then it selects an edge (it points to the other node in that edge) and both the nodes in that edge turn in; thus it straightforwardly follows the standard sequential 2-approximation algorithm. If some node k (at a distance farther than 3 from i) executes and selects $k' \in Adj_k$ to turn in, then and neither i nor j can be a neighbour of k or k' . Thus any race condition is prevented. This shows that Algorithm 5.8 preserves the 2-approximability of the standard 2-approximation algorithm for VC. \square

Example 5.3. In Figure 5.5 (a), we show a graph (containing eight nodes v_1, \dots, v_8) and the lattice induced by Algorithm 5.8 in the state space of that graph. We are omitting how the value of $i[done]$ gets modified; we only show how the vertex cover is formed. Only the reachable states are shown. Each node in the lattice represents a tuple of states of all nodes $\langle v_1[st], v_2[st], \dots, v_8[st] \rangle$. \square

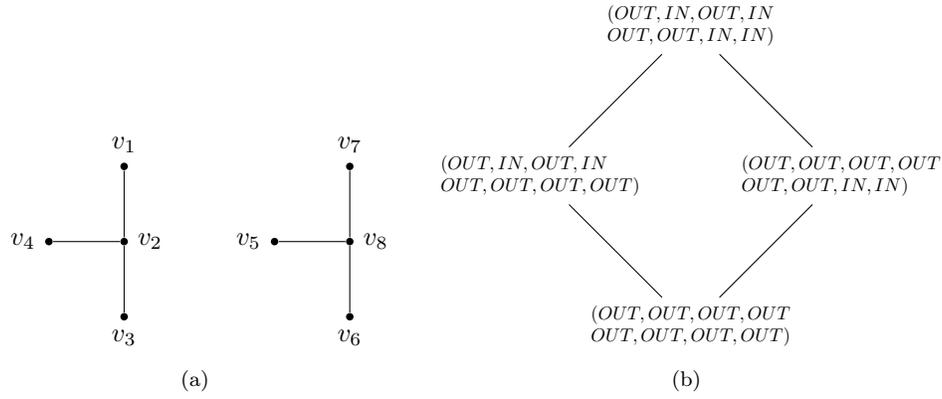


Figure 5.5: Execution of Algorithm 5.8: (a) input graph, and (b) lattice induced in the input graph. Transitive edges are not shown for brevity.

In Algorithm 5.8, local state of any node i is represented by two variables $i[done]$ and $i[st]$. Observe that in this algorithm, the definition of a node being impedensable depends on $i[done]$ and not $i[st]$. Therefore the transitions and consequently the structure of the lattice depends on $i[done]$ only, whose domain is of size 2.

In this algorithm, a node i makes changes to the variables of another node j , which is, in general, not allowed in a distributed system. We observe that this algorithm can be transformed into a lattice-linear distributed system algorithm where any node only makes changes only to its own variables. We describe the transformed algorithm, next.

5.7.2 Distributed Version of Algorithm 5.8

In Algorithm 5.8, we presented a lattice-linear 2-approximation algorithm for VC. In that algorithm, the states of two nodes i and j were changed in the same action.

Here, we present a mapping of that algorithm where i and j change their states separately. The key idea of this algorithm is when i intends to add j to the VC, $i[point]$ is set to j . When i is pointing to j , j has to execute and add itself to the VC. Thus, the transformed algorithm is as shown in Algorithm 5.9.

Algorithm 5.9. *Algorithm 5.8 transformed where every node modifies only its own variables.*

Init: $\forall i \in V(G), i[st] = OUT, i[done] = false, i[point] = \top$.

ELSE-POINTED-VC2A(i) $\equiv \exists j \in Adj_i^4 : \exists k \in Adj_j :$
 $k[point] = j \wedge j[done] = false$.

IMPEDENSABLE-VC2A-II(i) \equiv
 $(i[done] = false \wedge (\exists j \in Adj_i : j[point] = i) \vee$
 $(\neg \text{ELSE-POINTED-VC2A}(i) \wedge$
 $(i[done] = false \wedge (\forall j \in Adj_i^3 : j[id] < i[id] \vee$
 $j[done] = true)))$.

Rules for node i :

IMPEDENSABLE-VC2A-II(i) \longrightarrow

if $(\exists j \in Adj_i : j[point] = i)$, then

if $(\forall k \in Adj_i : k[st] = IN)$, then $i[done] = true$

else, then $i[st] = IN, i[done] = true$.

else, then

if $(\forall k \in Adj_i, k[st] = IN)$, then $i[done] = true$.

else, then

$j = \arg \max\{x[id] : x \in Adj_i \wedge x[done] = false\}$.

$i[st] = IN$.

$i[point] = j$.

$i[done] = true$.

Remark: Observe that in this algorithm, any j chosen throughout the algorithm by some impedensable i does not move in if it is already covered. Thus, we have that this algorithm computes a minimal as well

as a 2-approximate vertex cover. On the other hand, Algorithm 5.8 is a faithful replication of the classic sequential 2-approximation algorithm for vertex cover.

5.8 Gathering Robots on Triangular Grid

In this section, we study the problem of gathering distance-1 myopic robots on an infinite triangular grid. We show that the algorithm developed by Goswami et al. [16] is lattice-linear. Hence, this algorithm will run correctly even if the robots run in asynchrony where the robots can execute on old information about other robots. Because of lattice-linearity, this algorithm works correctly even if the robots are equipped with a unidirectional *camera* to see neighbouring robots. Authors of [16] assumed a distributed scheduler, which would require an omnidirectional camera, capable to get fresh values from all neighbouring locations.

Lattice-linearity follows that the moves of the robots are predictable. This allows us to show tighter bounds to the arena traversed by the robots under the algorithm. As a consequence of tighter bounds on this arena, (1) we obtain a better convergence time bound for this algorithm, which is lower than that showed in [16], and (2) we show that the gathering point of the robots can be uniquely determined from the initial, or any intermediate, state. We show that this algorithm converges in $2n$ rounds, which is lower as compared to the time complexity bound ($2.5(n + 1)$ rounds) shown in [16].

5.8.1 Gathering of Distance-1 Myopic Robots on Infinite Triangular Grid (GMRIT)

This chapter focuses on the problem where the input is a swarm of robots with minimal capabilities. Each robot is present at a vertex on an infinite triangular grid. In the initial global state, the robots form a connected graph on the underlying grid. The robots agree on an axis (i.e. a direction and its orientation). The robots can move only across one edge at a time. Each robot is myopic, i.e., it can only sense if another robot is present at an adjacent vertex. Robots do not have an ability to *communicate* with each other. Under these constraints, it is required that all robots gather at one point.

Problem Statement

The input is a global state s that describes the location of n robots placed on the grid points of an infinite triangular grid G such that the robots form a connected graph. The GMRIT problem requires that all robots gather at one vertex of G and stay forever at that vertex subject to the following constraints:

- *Visibility:* A robot can only determine if another robot is present in a neighbouring location. It cannot exchange data with another robot.
- *One Axis Agreement:* All robots agree on one axis and the orientation of that axis. (cf. y -axis as shown in Figure 5.6).

All robots are independent and identical from a physical and computational perspective and do not have an ID. They are oblivious to the coordinates of their location on the infinite triangular grid G . Observe that we can allow a global state s to be a multiset of vertices, each of which is the location of a robot. For s , its

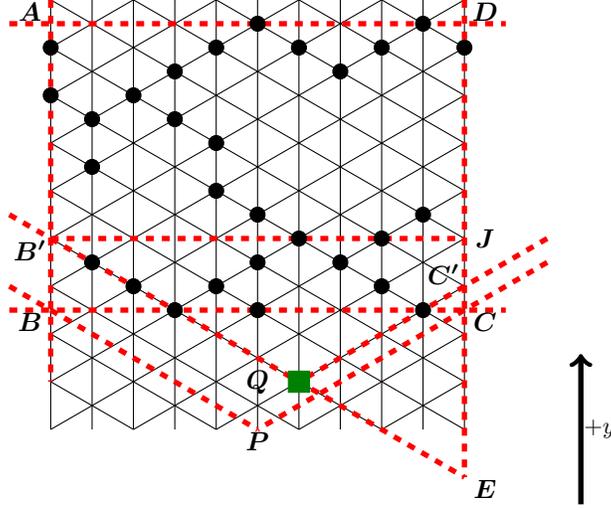


Figure 5.6: Robots on an infinite triangular grid: one on every round highlighted vertex.

visibility graph is the subgraph of G induced by the set of vertices in s .

Notice that by definition of the problem statement, a solution to GMRIT will provide silent self-stabilization. An instance of GMRIT is shown in Figure 5.6. Here, each round highlighted vertex represents a robot. Observe that the visibility graph of this global state is a connected graph.

Next, we discuss how GMRIT problem is not a lattice-linear problem. This can be illustrated by a system containing two robots x_1 and x_2 present at locations l_1 and l_2 (l_1 and l_2 are different vertices on the same edge) on G . In such a system x_1 can move to l_2 , in which case, x_2 is not impendensable, or otherwise, x_2 can move to l_1 , in which case, x_1 is not impendensable. Hence, no specific robot can be deemed impendensable, though, the global state is suboptimal.

Problem Specific Definitions

Some of the definitions that we discuss in this subsection are from [16]. A *horizontal layer* is a line perpendicular to the y -axis that passes through at least one robot. The *top layer* of a global state s is a horizontal layer such that there is no horizontal layer above it (e.g., AD in Figure 5.6). *Bottom layer* of s is a horizontal layer such that there is no horizontal layer below it (e.g., BC in Figure 5.6).

A *vertical layer* is parallel to the y -axis such that it passes through at least one robot. The *left layer* of s is the vertical layer such that there is no vertical layer on its left (e.g., AB in Figure 5.6). The *right layer* of s is the vertical layer such that there is no vertical layer on its right (e.g., CD in Figure 5.6).

As seen in Figure 5.6, vertices in G are intersections of three groups of parallel lines; one of these groups are lines parallel to the y -axis. We use p -axis (positive slope) and n axis (negative slope) to denote the other group of parallel lines. The *positive slant* is a line parallel to p -axis (e.g., BP in Figure 5.6) and *negative slant* is a line parallel to n -axis (e.g., CP in Figure 5.6). The *bottom l2r slant* of s is a negative slant that

passes through a robot such that there is no negative slant on its left passing through a robot (e.g., $B'Q$ in Figure 5.6). The *bottom r2l slant* of s is a positive slant that passes through a robot such that there is no positive slant on its right passing through a robot (e.g., $C'Q$ in Figure 5.6). Note that a negative slant and a positive slant can be imaginary, or a line in G .

The *depth* of s is the distance between its top layer and its bottom layer. The *width* of s is defined as the distance between its left layer and right layer.

As shown in Figure 5.6, a polygon $ABPCD$ is a *bounding polygon* of a global state s if (1) AB and CD are line segments of the left layer and the right layer of s respectively, (2) AD and BC are line segments of the top layer and the bottom layer of s respectively, and (3) P is the point of intersection between the negative slant passing through B and the positive slant passing through C .

Note that these definitions (top/bottom layer, etc.) are only used for discussion of the protocol and proofs. The robots are not aware of them. Similarly, the robots can distinguish between *up* and *down*, but not *left* and *right*.

5.8.2 General Idea of the Algorithm

A robot has six possible neighbouring locations. The naming convention for these locations is as shown in Figure 5.7 (a) [16]. Since each robot has 6 neighbouring locations, it can be in one of 2^6 possible local states. Of these, the robot can move in only 11 states. Of these, 7 states are shown in Figure 5.7 (b) [16]. The other 4 states are mirror images of those shown in cases 2, 5, 6 and 7 in Figure 5.7 (b).

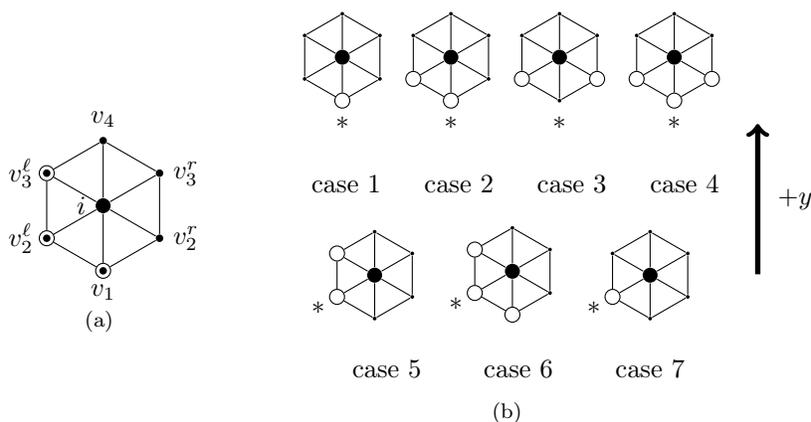


Figure 5.7: (a) Naming conventions for neighbourhood of a robot. (b) Cases where a node is impedensable. Note that the mirror images of these local states are also impedensable.

Authors of [16] show that the robots do not move out of the bounding polygon. They also show that the visibility graph induced among the robots stay connected, and the dimensions of the bounding polygon reduce with every round.

5.8.3 GSGS Algorithm [16] for GMRIT problem

In this section, we reword the algorithm in [16] to demonstrate its lattice-linearity. We define the macros listed in Table 5.4. For a set L of locations around a node i , $\text{AT}(i, L)$ is true iff if there is at least one robot at each location in L . $\text{ONLY-AT}(i, L)$ is true iff $\text{AT}(i, L)$, and there is no other robot at locations other than locations in L . A robot i is *extreme* if (1) there is no robot on *top* ($v_4(i)$) of i and (2) if there is a robot on the left (v_2^ℓ or v_3^ℓ) of i , then there is no robot on the right (v_2^r or v_3^r) of i . If a robot i is extreme, and there is no robot around it, then i is a *terminating* robot. If i is extreme, and there is a robot only on $v_3(i)$ or there are robots only on both $v_1(i)$ and $v_3(i)$, then i is a *staying* robot.

If i is extreme, and there is a robot on $v_1(i)$ and no robot on $v_3(i)$, then i is a *downward impedensable* robot. If i is not a downward impedensable robot, not a staying robot, and not a terminating robot, then it is a *downslant impedensable* robot. If i is not an extreme robot, and there is a robot on both $v_2(i)$ and no robot at its y-coordinate > 0 , then i is a *non-extreme impedensable* robot. The algorithm is described as follows.

| |
|---|
| $\text{EXTREME}(i) \equiv \neg \text{AT}(i, \{v_4\}) \wedge ((\text{AT}(i, \{v_2^r\}) \vee \text{AT}(i, \{v_3^r\})) \Rightarrow (\neg \text{AT}(i, \{v_2^\ell\}) \wedge \neg \text{AT}(i, \{v_3^\ell\})))$ $\text{TERMINATING}(r) \equiv \text{EXTREME}(i) \wedge (\forall q \in \{v_1, v_2^r, v_3^r, v_4, v_3^\ell, v_2^\ell\} : \neg \text{AT}(i, \{q\})).$ $\text{STAYING}(i) \equiv \text{EXTREME}(i) \wedge (\text{ONLY-AT}(i, \{v_3^r\}) \vee \text{ONLY-AT}(i, \{v_3^\ell\}) \vee \text{ONLY-AT}(i, \{v_1, v_3^r\}) \vee \text{ONLY-AT}(i, \{v_1, v_3^\ell\})).$ $\text{DOWNWARD}(i) \equiv \text{EXTREME}(i) \wedge \text{AT}(i, \{v_1\}) \wedge \neg(\text{AT}(i, \{v_3^r\}) \vee \text{AT}(i, \{v_3^\ell\})).$ $\text{DOWNSLANT-RIGHT}(i) \equiv \text{EXTREME}(i) \wedge \neg \text{DOWNWARD}(i) \wedge \neg \text{STAYING}(i) \wedge \neg \text{TERMINATING}(r) \wedge \text{AT}(i, \{v_2^r\}).$ $\text{DOWNSLANT-LEFT}(i) \equiv \text{EXTREME}(i) \wedge \neg \text{DOWNWARD}(i) \wedge \neg \text{STAYING}(i) \wedge \neg \text{TERMINATING}(r) \wedge \text{AT}(i, \{v_2^\ell\}).$ $\text{NON-EXTREME}(i) \equiv \neg \text{EXTREME}(i) \wedge \text{AT}(i, \{v_2^r, v_2^\ell\}) \wedge \neg(\text{AT}(i, \{v_3^r\}) \vee \text{AT}(i, \{v_3^\ell\}) \vee \text{AT}(i, \{v_4\})).$ $\text{IMPEDENSABLE-GSGS}(i) \equiv \text{DOWNWARD}(i) \vee \text{DOWNSLANT-RIGHT}(i) \vee \text{DOWNSLANT-LEFT}(i) \vee \text{NON-EXTREME}(i).$ |
|---|

Table 5.4: Macros used in the algorithm for GMRIT problem.

If a robot i is *downward impedensable*, then i moves downwards to $v_1(i)$. If i is *downslant impedensable*, then i moves to $v_2(i)$. If i is a *non-extreme impedensable* robot, then i moves to $v_1(i)$.

Algorithm 5.10. *Rules for robot i .*

$$\boxed{\text{IMPEDENSABLE-GSGS}(i) \longrightarrow \begin{cases} \text{move}(i, v_1(i)) & \text{if DOWNWARD}(i) \\ \text{move}(i, v_2^r(i)) & \text{if DOWNSLANT-RIGHT}(i) \\ \text{move}(i, v_2^l(i)) & \text{if DOWNSLANT-LEFT}(i) \\ \text{move}(i, v_1(i)) & \text{if NON-EXTREME}(i) \end{cases}}$$

In [16], authors assume a distributed scheduler. Next, we show that Algorithm 5.10 is lattice-linear, Thus, it will be correct even in asynchrony.

Lattice-Linearity

In this subsection, we show lattice-linearity of Algorithm 5.10. Among the lemmas and theorems presented here, Lemma 5.7 and Lemma 5.8 are adopted from [16]. We use them to help prove some properties of Algorithm 5.10. All other results show or arise from the lattice-linearity of Algorithm 5.10.

Lemma 5.5. *The predicate $\forall i : \neg \text{IMPEDENSABLE-GSGS}(i)$ is a lattice-linear predicate on n robots, and the visibility graph does not get disconnected by the actions under Algorithm 5.10.*

Proof. In this proof, we consider the 7 cases as shown in Figure 5.7 (b) and show that if robot i is impedensable, it must execute to reach the goal state. We show that if a robot i is impedensable, then there exists at least one robot j around i which does not move until i moves. Specifically, Algorithm 5.10 imposes that j ‘waits’ for i to move. It means that if i does not move, then the robots cannot find the gathering point.

Case 1: This robot i is a downward impedensable robot. The other robot that is present below it is not extreme and is also not a non-extreme impedensable robot because i is present above it, so it will not move until i changes its location.

Case 2: This robot i is a downward impedensable robot. There are two robots, x_1 and x_2 , present at locations $v_1(i)$ and $v_2(i)$ respectively. x_1 is not extreme and is also not non-extreme impedensable because i is present above it. So x_1 will not move until i changes its location. x_2 may be impedensable. x_2 can only move to the location of x_1 thereby resulting in case 1. In this possibility, the robot i remains impedensable and its required action does not change.

Case 3: This robot i is a non-extreme impedensable robot. There are two robots, x_1 and x_2 , present at locations $v_2(i)$ -left and $v_2(i)$ -right respectively. For x_1 or x_2 to be impedensable, there must be some robot at the $v_1(i)$ location, which is not the case, thus, they are not impedensable. Hence, x_1 and x_2 will not move until i changes its location.

Case 4: This robot i is a non-extreme impedensable robot. There are three robots, x_1 , x_2 and x_3 , present at locations $v_2(i)$ -left, $v_1(i)$ and $v_2(i)$ -right respectively. x_2 is not impedensable. x_1 and x_3 may be

impedensable, based on their local states. If one or both of them move, they will move to the location of x_2 , resulting in case 2 or case 1. In these possibilities, the robot i remains impedensable and its required action does not change.

Case 5: This robot i is a downslant impedensable robot. There are two robots, x_1 and x_2 , present at locations $v_2(i)$ and $v_3(i)$ respectively. x_1 is not extreme and is also not non-extreme impedensable. So x_1 will not move until i changes its location. x_2 may be impedensable, based on its local state. x_2 can move to the location of x_1 or i thereby resulting in case 7. In this possibility, the robot i remains impedensable and its required action does not change.

Case 6: This robot i is a downslant impedensable robot. There are three robots, x_1 , x_2 and x_3 , present at locations $v_1(i)$, $v_2(i)$ and $v_3(i)$ respectively. x_1 and x_2 are not extreme and are also not non-extreme impedensable. Initially, x_1 and x_2 cannot move. x_3 may be downward impedensable, based on its local state. x_3 can only move to the location of x_2 thereby resulting in case 2. After this, one or both of x_1 and x_2 can move to the location of x_1 , resulting in case 1. In these possibilities, i remains impedensable, its required action may change, but the graph does not get disconnected if it executes under case 6 (using old information, if it assumes, despite the movement of other robots, that it falls in case 6).

Case 7: This robot i is a downslant impedensable robot. The other robot x_1 that is present at $v_2(i)$ is not extreme and is also not a non-extreme impedensable robot, so x_1 will not move until i changes its location.

From these cases, we also have that an impedensable robot stays connected to the robots that were its neighbours before it moved. This implies that the visibility graph stays connected after any impedensable robots move. □

The robots executing Algorithm 5.10, as shown in [16], stay in the bounding polygon $ABPCD$. Next, we show, using the above proof, a tighter polygon bounding the robots. To define this polygon, we let Q to be the point such that it is an intersection between the bottom $l2r$ slant of s and the bottom $r2l$ slant of s (cf. Figure 5.6). Let B' be the point of intersection between left layer (AB) and the bottom $l2r$ slant of s and let C' be the point of intersection between the right layer (CD) and the bottom $r2l$ slant of s . We show that the robots never step out of the polygon $AB'QC'D$, which is tighter than $ABPCD$.

Observation 5.1. *If the neighbouring robot, say j of an impedensable robot i moves then i or j fall under case 5 or case 6.*

Lemma 5.6. *Throughout the execution of Algorithm 5.10, the bottom $r2l$ slant and the bottom $l2r$ slant will not change.*

Proof. In a global state s , a robot present at the bottom $l2r$ slant of s or a bottom $r2l$ slant of s is represented in cases 5 and 7. From Algorithm 5.10 and the proof of Lemma 5.5, if a robot is present at bottom $l2r$ slant (respectively, bottom $r2l$ slant), it will never move below $(v_1(i))$ or left $(v_2^l(i))$ of its location (respectively, below $(v_1(i))$ or right $(v_2^r(i))$ of its location). \square

Lemma 5.7. [16] *Throughout the execution of Algorithm 5.10, left layer does not move leftwards and right layer does not move rightwards.*

Lemma 5.8. [16] *In every round of Algorithm 5.10, the top layer moves at least $1/2$ unit in the negative direction of the y -axis.*

Corollary 5.1. (From Lemma 5.6 and Lemma 5.7) *The robots will never step out of the polygon $AB'QC'D$.*

Theorem 5.6. *Algorithm 5.10 is a lattice-linear self-stabilizing algorithm for the GMRIT problem on n robots executing asynchronously.*

Proof. From Lemma 5.6, we have that bottom $l2r$ slant and bottom $r2l$ slant do not change. From Corollary 5.1, we have that the robots will never step out of the polygon $AB'C'DQ$. From Lemma 5.8, we have that the top layer moves down by at least half a unit in the negative direction of y -axis. Thus we have that the robots converge at the point of intersection of the bottom $l2r$ slant and the bottom $r2l$ slant, and the robots will eventually gather at that point. \square

Corollary 5.2. (From Lemma 5.8, Corollary 5.1 and Theorem 5.6) *The point, Q , where the robots gather, can be uniquely determined from the initial global state.*

Time Complexity Properties

In [16], authors showed that Algorithm 5.10 converges in $2.5(n+1)$ rounds. Based on Corollary 5.2 which identifies a predictable gathering point, we show that a maximum of $2n$ rounds is sufficient, which is a tighter bound.

Theorem 5.7. *Algorithm 5.10 converges in $2n$ rounds.*

Proof. We use Figure 5.6 to discuss convergence of robots in Algorithm 5.10. As discussed in Section 5.8.1 and Section 5.8.3, let A , B' , Q , C' and D be the points obtained by pairwise intersection of the top layer, left layer, bottom $l2r$ slant, bottom $r2l$ slant and right layer. Let h_ℓ be the depth of the line segment AB' , h_r be the depth of the line segment $C'D$, and w be the width of the line segment AD . Note that a unit of length of the depth of AB' or $C'D$ is $\sqrt{3}$ times a unit of length of the width of AD due to the geometry of G . Since the robots form a connected graph, $w \leq n$. And, if $w > 0$ then $h_\ell + h_r \leq n$. If $w = 0$, we define $h_\ell = 0$ and $h_r = n$.

In the case where $w = 0$, it can be clearly observed that Algorithm 5.10 converges in n rounds. Next, we consider if $w > 0$. Without the loss of generality, let $h_r \geq h_\ell$. Thus, $h_\ell \leq n/2$.

Let E be the point of intersection between the bottom $l2r$ slant and the right layer (cf. Figure 5.6). We draw a horizontal line (perpendicular to the y -axis) through B' and use J to denote its intersection with DC' . Thus, the depth of $AB'JD$ is h_ℓ . Additionally, observe that the length of $B'E$ on the n -axis is w . Thus the height of JE is $w/2$ units on the y -axis. This means that the depth of $B'EJ$ is $w/2$. By construction of E , the depth of $B'QC'J$ is upper bounded by the depth of $B'EJ$. Thus, the depth of $B'QC'J$ cannot exceed $w/2 \leq n/2$ units.

Thus, the total depth of $AB'QC'D$ is equal to the sum of the depth of $AB'JD$ and the depth of $B'QC'J$, which is upper bounded by $n/2 + n/2 = n$ units. From Lemma 5.8, the total number of rounds required for the robots to gather is upper bounded by $2n$ moves. \square

5.8.4 Revised Algorithm for GMRIT

In this section, we present a revised algorithm that simplifies the proof of lattice-linearity. This algorithm is based on the difficulties involved in the proof of Lemma 5.5 where we needed to consider the possible actions taken by the neighbours of an impedensable robot. Our proof would have been simpler if all the neighbours of an impedensable robot i would not be allowed to move until i moves. Additionally, from Observation 5.1, we have that if a robot j , neighbouring to an impedensable node i , is impedensable, then i or j fall in case 5 or case 6.

These issues can be alleviated by removing cases 5 and 6 from the algorithm. The macros that we utilize are as follows. A robot is *downward impedensable* if its local state is one of those represented in cases 1, 2, 3 or 4 (and their mirror images; cf Figure 5.7 (b)). A robot is *downslant impedensable* if its local state is that represented in case 7.

$$\text{DOWNWARD-II}(i) \equiv (\text{AT}(i, \{v_1\}) \wedge \neg \text{AT}(i, \{v_3^\ell, v_4, v_3^r\})) \vee \text{ONLY-AT}(i, \{v_2^\ell, v_2^r\}).$$

$$\text{DOWNSLANT-LEFT-II}(i) \equiv \text{ONLY-AT}(i, \{v_2^\ell\}).$$

$$\text{DOWNSLANT-RIGHT-II}(i) \equiv \text{ONLY-AT}(i, \{v_2^r\}).$$

The revised algorithm is as follows. A downward impedensable robot moves to $v_1(i)$ location, and a downslant impedensable robot moves to $v_2(i)$ location.

Algorithm 5.11. *Rules for robot i .*

| |
|--|
| $\text{DOWNWARD-II}(i) \longrightarrow \text{move}(i, v_1(i)).$ |
| $\text{DOWNSLANT-RIGHT-II}(i) \longrightarrow \text{move}(i, v_2^r(i)).$ |
| $\text{DOWNSLANT-LEFT-II}(i) \longrightarrow \text{move}(i, v_2^\ell(i)).$ |

In Algorithm 5.11, because of the removal of cases 5 and 6, any robot around an impedensable robot does not move. Thus lattice-linearity of this algorithm can be visualized more intuitively. Consequently, we have the following lemma.

Lemma 5.9. *The predicate $\forall i : \neg(\text{DOWNWARD-II}(i) \vee \text{DOWNSLANT-RIGHT-II}(i) \vee \text{DOWNSLANT-LEFT-II}(i))$, is a lattice-linear predicate on n robots, and the visibility graph does not get disconnected by the actions under Algorithm 5.11.*

Lemma 5.8 shows the top-layer moves down in each round. This proof is not affected by the removal of cases 5 and 6, as the robot executing in cases 5 or 6 is *not* a top-layer robot. Consequently, Algorithm 5.11 follows the properties as described in Lemma 5.8, Theorem 5.6 and hence Theorem 5.7. Thus, we obtain the following theorem.

Theorem 5.8. *Algorithm 5.11 is a lattice-linear self-stabilizing algorithm for GMRIT problem on n robots executing asynchronously. It converges in $2n$ rounds, and the robots gather at Q .*

5.9 Summary of the chapter

In this chapter, we introduced *fully lattice-linear algorithms* that are tolerant to asynchrony. Such algorithms induce lattices in the state space even if the underlying problem does not specify, in a suboptimal global state, a set of nodes that must change their states.

5.9.1 Theoretical Achievements

We bridge the gap between lattice-linear problems [6] and eventually lattice-linear algorithms (Chapter 4). Fully lattice-linear algorithms overcome the limitations of [6] and Chapter 4. Additionally, such algorithms can be developed even for problems that are not lattice-linear. This overcomes a key limitation of [6] where the system fails if nodes cannot be deemed impedensable, or not impedensable, naturally. Since the lattice structures exist in the entire (reachable) state space, we overcome a limitation of Chapter 4 where only in a subset of global states, lattice-linearity is observed.

5.9.2 New Fully Lattice-Linear Algorithms

We present algorithms for minimal dominating set (MDS), graph colouring (GC), minimal vertex cover (MVC) and maximal independent set (MIS). Of these, MDS and GC relied on tie-breakers on node IDs, a common approach for breaking ties in the literature. We observe that a similar design cannot be directly extended to develop an algorithm for MVC and MIS. However, the use of complex actions – that permit a node to change the values of the variables of other node as well as its own – enable the design of algorithms for MVC and MIS. We also observe that complex actions can be revised into simple actions – where a node can only change its own values – without losing lattice-linearity. However, these revised algorithms utilize the phenomenon of priority inheritance to accomplish this.

We also provide a fully lattice-linear 2-approximation algorithm for vertex cover. This algorithm is the first lattice-linear approximation algorithm for an NP-Complete problem.

In [6] lattice-linearity is studied in only those systems where the state space forms a distributive lattice where all pairs of global states have a join (supremum) and meet (infimum), and join and meet operations distribute over each other. We observe that some of these requirements are not required to provide correctness under asynchrony. Specifically, we observe that a system allows asynchrony if the state space forms \prec -lattices, where the join between any two states is defined, but the definition of meet is not required. This aspect is more overtly observed in instances of MVC. Specifically, Figure 5.3 shows that we have a \prec -lattice but not a distributive lattice.

Fully lattice-linear algorithms considered in this chapter preserve an advantage of [6] that was lost in the extension by Chapter 4. Specifically, in [6], the final configuration could be uniquely determined from the initial state, whereas in Chapter 4, all global states (specifically, the infeasible states) do not form a lattice, so starting from an arbitrary state, the state of convergence cannot be predicted. In fully lattice-linear algorithms that we introduce in this chapter, the state space is split into multiple lattices and the algorithm starts in one of them. Hence, the state of convergence can be uniquely determined by the initial state.

5.9.3 Distance-1 Transformation and Experiments

We have that a lattice-linear algorithm can be transformed to a distance-1 algorithm by having the nodes keep a copy of the variables, of the other nodes, that they want to evaluate their guards with. We transform Algorithm 5.2 to a distance-1 algorithm by using a minimal set of variables needed to evaluate said guards.

We also demonstrate that these algorithms substantially benefit from using the fact that they satisfy the property of lattice-linearity. Specifically, they outperform existing algorithms when they utilize the fact that they are correct without synchronization among processes, i.e., they are correct even if a node is reading old/inconsistent values of its neighbours.

5.9.4 Gathering Myopic Robots

We also show lattice-linearity of the algorithm developed by Goswami et al [16] for gathering robots on an infinite triangular grid. This removes the assumptions of synchronization from the algorithm and thus makes a system running this algorithm fully tolerant to asynchrony. We also present a revised algorithm that simplifies the proof of lattice-linearity without losing any of the desired properties (e.g., convergence time, stabilization).

Lattice-linearity implies that the locations, possibly visited by a robot, form a total order. The total order is a result of the fact that we are able to determine all and the only robots in any global state that are impedensable, and an impedensable robot has only one choice of action. By making this observation, it can also be noticed that we can closely predict the executions that the robots would perform. As a

result, we are able to (1) compute the exact arena traversed by the robots throughout the execution of the algorithm (Lemma 5.6 and Lemma 5.7), and (2) deterministically predict the point of gathering of the robots (Corollary 5.2).

We also provided a better upper bound on the time complexity of this algorithm. Specifically, we show that it converges in $2n$ rounds, whereas [16] showed that a maximum of $2.5(n+1)$ rounds are required. This was possible due to the observations that followed from the proof of lattice-linearity of this algorithm.

CHAPTER 6

PARTIAL ORDER-INDUCING SYSTEMS

In the previous chapters, we explored several special cases of algorithms that can converge without synchronization. Specifically, we studied how algorithms, that impose the nodes to visit their local states in a total order, tolerate asynchrony. Consequently, the global state transition graph forms a \prec -lattice, and thus, we call such systems lattice-linear systems. We also studied several example problems and algorithms that act as concrete evidence of our theory. However, as we will study in this chapter, an arbitrary system, that can converge in asynchrony, may not be lattice-linear. Lattice induction is a sufficient condition to allow asynchrony, but it is not a necessary condition. In this chapter, we complete the theory that explains the behaviour of an arbitrary system that can tolerate asynchrony. To this end, we introduce partial order-inducing problems and partial order-inducing algorithms.

We show that induction of a \prec -DAG (induced among the global states – that forms as a result of a partial order induced among the local states visited by individual nodes) is a necessary and sufficient condition to allow an algorithm to run in asynchrony.

In the chapter, we first provide a comprehensive description of partial order-inducing problems and partial order-inducing algorithms, along with some simple examples. Then we show some properties of an algorithm that can converge under asynchrony, which include the condition that we discussed in the above paragraph.

An important conclusion from the above observation is that if we want to show that an algorithm can converge without synchronization, then we do not have to generate the entire global state transition system and check for the absence of cycles. Rather, we only need to show that the local state transition graph forms a partial order (PO). Thus, the complexity of determining the correctness of such systems is significantly reduced, and so this observation is fruitful in writing social and formal proofs that show tolerance of an algorithm to asynchrony.

In this chapter, we study problems such as the dominant clique (DC) problem, the shortest path (SP) problem and the maximal matching (MM) problem. We show that DC and SP are PO-inducing problems. Among these, DC allows self-stabilization, whereas the algorithm that we present for the SP does not. We demonstrate that MM is not a PO-inducing problem. We present a PO-inducing algorithm for it. This algorithm allows self-stabilization. We study the upper bound to the convergence time of a PO-inducing algorithm. We show how inducing a partial order among the local states of all individual nodes is crucial to allow asynchrony: it is necessary and sufficient condition to allow asynchrony.

A observation that immediately follows is that since a total order is a special case of a partial order, all lattice-linear problems and algorithms are, respectively, PO-inducing problems and algorithms.

Organization of the Chapter

This chapter is organized as follows. In Section 6.1, we study the characteristics of PO-inducing problems, and in Section 6.2, we study the characteristics of PO-inducing algorithms, with examples. In Section 6.3, we study the properties of PO-inducing algorithms. While the previous sections provide simple, but sufficient, examples of asynchrony tolerant systems, this section is crucial from the perspective of the theory that we establish in this chapter. Finally, we summarize the chapter in Section 6.4.

6.1 Natural partial order induction: PO-inducing Problems

In this section, we discuss properties of problems where a partial order (PO) among the nodes visited by individual nodes is induced *naturally*. It means that in any suboptimal state, the problem definition itself specifies the nodes that must change their state, in order for the system to reach an optimal state.

6.1.1 Embedding a \prec -DAG among global states

To explain the embedding of a \prec -DAG, first, we define a partial order \prec_l among the local states of a node. This partial order defines all the possible transitions that a node is allowed to take. The partial order \prec_l is used to restrict how node i can execute: i can go from state $s[i]$ to $s'[i]$ only if $s[i] \prec_l s'[i]$.

Using \prec_l , we define \prec_g that orders the global states. The predicate $s \prec_g s'$ is true iff the predicate $(\forall i : s[i] = s'[i] \vee s[i] \prec_l s'[i]) \wedge (\exists i : s[i] \prec_l s'[i])$ is true; $s = s'$ iff $\forall i : s[i] = s'[i]$. For brevity, we use \prec to denote \prec_l and \prec_g : \prec corresponds to \prec_l while comparing local states, and \prec corresponds to \prec_g while comparing global states. We also use the symbol ' \succ ' which is such that $s \succ s'$ iff $s' \prec s$. Similarly, we use symbols ' \preceq ' and ' \succeq '; e.g., $s \preceq s'$ iff $s = s' \vee s \prec s'$. We call the DAG, formed from such partial order, a \prec -DAG.

Definition 6.1. \prec -DAG. *Given a partial order \prec_l that orders the local states visited by i (for each i), the \prec -DAG corresponding to \prec_l is defined as follows: $s \prec s'$ iff $(\forall i : s[i] \preceq_l s'[i]) \wedge (\exists i : s[i] \prec_l s'[i])$.*

A \prec -DAG constraints how global states can transition among one another: state s can transition to state s' iff $s \prec s'$. By varying \prec_l that identifies a partial order among the local states of a node, one can obtain different \prec -DAGs. A \prec -DAG, embedded in the state space, is useful for permitting the algorithm to execute asynchronously. Under proper constraints on the structure of \prec -DAG, convergence can be ensured. We elaborate on this in Section 6.1.2.

6.1.2 General Properties of PO-Inducing Problems

Asynchrony can be allowed in any system that imposes a condition that any node i changes its state only if it evaluates that an optimal global state cannot be reached with the current local state of i . We call such a node an *impedensable node* (*indispensable* to change for progress, an *impediment* to progress if it does not change its state). Let \mathcal{P} be the predicate governing a system such that it determines the state transitions of that system, that is, it determines which nodes have to change their state. And, let $\mathcal{P}(s)$ be true only if no

nodes in global state s are impedensable.

Impedensable Node and Impedensable Global State

In a multiprocessor system, a node can have single or multiple choices of action when it changes its state, but it can be in only one state at a given instance. A node i is enabled iff it is impedensable. In this case, (1) the nodes are also capable of discarding more than one local state, and (2) consequently, a partial order is induced among the local states visited by individual nodes.

Let i be impedensable in a global state s , and let st be the current local state of i . Node i does not revisit st , and it also does not visit some other local states. To explain this, let $EQUICORRUPT(st, i)$ be the set of local states of i that are deemed to be violating iff state st is a violating local state in i . So if st is discarded, all states in $EQUICORRUPT(st, i)$ are also not visited by i throughout the execution. This ensures that i must move up in the partial order. We have the definition of an impedensable node as follows.

Definition 6.2. Impedensable node (updated definition). $IMPEDENSABLE(i, s, \mathcal{P}) \equiv \neg \mathcal{P}(s) \wedge (\forall s' : (s' \succeq s) \Rightarrow (s'[i] \in EQUICORRUPT(s[i], i) \Rightarrow \neg \mathcal{P}(s')))$.

Example 6.Max: Continuation 6. Consider the execution of the nodes in global state $\langle 2, 2, 3 \rangle$ under the algorithm for the max problem presented in Section 2.1.3. $EQUICORRUPT(2, 1)$ is the set of local states of node 1 that are equally corrupt as local state 2. Due to the total order, there is only one node in each level, and so, e.g., the set $EQUICORRUPT(2, 1) = \{2\}$ has only one value. \square

As we will study in Section 6.1.3 and in the following parts of the chapter, the states in $EQUICORRUPT(st, i)$ are essentially the local states of i that are at the same level in the partial order. Thus, a global state s is suboptimal, that is, s is impedensable, if and only if it contains at least one impedensable node. Formally,

Definition 6.3. Impedensable global state. $IMPEDENSABLE(s, \mathcal{P}) \equiv \exists i : IMPEDENSABLE(i, s, \mathcal{P})$.

PO-inducing problems

A *PO-inducing problem* P can be represented by a predicate \mathcal{P} , where P stipulates that any local state st of i , that is deemed in violation by i , will make any global state s suboptimal if $s[i] = st$. So i never revisits st . As a result, predicate \mathcal{P} induces a partial order among the local states visited by a node, for all nodes (no cycles). Consequently, the discrete structure \mathcal{S} that gets induced among the global states is a \prec -DAG, as described in Definition 6.1. We say that \mathcal{P} , satisfying Definition 6.2, is *PO-inducing* with respect to that \prec -DAG. \mathcal{P} is used by the nodes to determine if they are impedensable, using Definition 6.2 and Definition 6.3.

Definition 6.4. PO-Inducing Predicate. \mathcal{P} is a *PO-inducing predicate* with respect to a \prec -DAG induced among the global states iff $\forall s \in S : \neg \mathcal{P}(s) \Rightarrow \exists i : IMPEDENSABLE(i, s, \mathcal{P})$.

Remark: Since a total order is a special case of a partial order, all lattice-linear problems are PO-inducing problems.

Now we complete the definition of PO-inducing problems. In a PO-inducing problem P , given any suboptimal global state, we can identify all nodes that should not retain their state. \mathcal{P} is thus designed conserving this nature of problem P .

Definition 6.5. PO-inducing problem (DIP). A problem P is PO-inducing iff there exists a predicate \mathcal{P} and a \prec -DAG \mathcal{S} , induced among the global states, such that (1) P is solved iff the system reaches a state where \mathcal{P} is true, (2) \mathcal{P} is PO-inducing with respect to \mathcal{S} , i.e., $\forall s : \neg\mathcal{P}(s) \Rightarrow \exists i : \text{IMPEDENSABLE}(i, s, \mathcal{P})$, and (3) $\forall s : (\forall i : \text{IMPEDENSABLE}(i, s, \mathcal{P}) \Rightarrow (\forall s' : \mathcal{P}(s') \Rightarrow s'[i] \neq s[i]))$.

Successors of Global States

When a system allows asynchrony, then a global state s can transition to one of multiple other global states. Let S_s be a set of such states. S_s will contain the states which s can transition to if all nodes are reading fresh local states of other nodes.

Definition 6.6. Successors of a global state. $\text{SUCCESSORS}(s) \equiv \{s' : s' \succ s\}$.

There may be some global states that do not have any successors. We call them *terminal successors*.

Definition 6.7. Terminal Successors. $s' \in \text{TERMINAL-SUCCESSORS}(s)$, iff $\{s' \mid s' \in \text{SUCCESSORS}(s) \wedge \text{SUCCESSORS}(s') = \phi\}$.

Example 6.Max: Continuation 7. Going back to Example 2.2 (Chapter 2), the only terminal successor in the state transition graph shown in Figure 2.1 (b) is $\langle 3, 3, 3 \rangle$. □

Self-Stabilizing Predicates

\mathcal{P} satisfies Definition 6.8 only if starting from any arbitrary state, the system converges to an optimal state. This, in turn, is possible only if all terminal successors in \mathcal{S} are optimal states. \mathcal{P} can be true in other states as well.

Definition 6.8. Self-stabilizing PO-inducing predicate. Continuing from Definition 6.5, \mathcal{P} is a self-stabilizing PO-inducing predicate if and only if all terminal successors in the \prec -DAG induced by \mathcal{P} are optimal states, i.e. $\forall s, s' \in S : \text{TERMINAL-SUCCESSOR}(s, s') \Rightarrow \mathcal{P}(s') = \text{true}$.

Example 6.Max: Continuation 8. We have shown an incomplete state transition graph in Figure 2.1 (b), however, it can be trivially noticed that the predicate for the max problem, as noted in Example Max: Continuation 4 is a self-stabilizing PO-inducing predicate. □

The main intent of this chapter is to establish some fundamental properties of PO-inducing systems. However, to understand these properties, we need to understand the behaviour of such systems. Thus, in the remaining part of this section, we study some PO-inducing problems, and explore how the induction of a partial order among the local states, under the acting algorithm, allows the nodes to execute without synchronization.

6.1.3 Dominant Clique (DC) Problem

In this section, we describe an algorithm for the dominant clique problem, which is defined in the following paragraph. The algorithm that we describe in this subsection is an *embarrassingly parallel algorithm*, i.e., in this algorithm, there is no transfer of data among the nodes. However, we use this trivial algorithm to build the intuition behind the induction of a partial order among the local states visited by individual nodes, and how that gives rise to a \prec -DAG among the global states.

Definition 6.9. Dominant Clique. *In the dominant clique problem, the input is an arbitrary graph G such that for the variable $i[cliq]$ of each node i , $i[cliq] \subseteq N_i$ and $\{i\} \subseteq i[cliq]$. The task is to (re-)evaluate $i[cliq]$ such that (1) all the nodes in $i[cliq]$ form a clique, and (2) there exists no clique c in G such that $i[cliq]$ is a proper subset of c .*

Thus, we define the DC problem by the following predicate.

$$\mathcal{P}_{dc} \equiv (i \in i[cliq]) \wedge (\forall j, k \in i[cliq] : (j \neq k) \Rightarrow (k \in Adj_j)) \wedge (\nexists j \in Adj_i : j \notin i[cliq] \wedge (\forall k \in i[cliq] : k \in Adj_j))$$

The local state of a node i is defined by $\langle i[cliq] \rangle$. An impedensable node i in a state s is a node for which (1) all the nodes in $i[cliq]$ do not form a clique, or otherwise (2) there exists some node k in Adj_i such that $i[cliq] \cup \{k\}$ is a valid clique, but k is not in $i[cliq]$. Formally,

$$\begin{aligned} \text{IMPEDENSABLE-DC}(i) \equiv & (i \notin i[cliq]) \vee \neg(\forall j, k \in i[cliq] : j \neq k \wedge j \in Adj_k) \vee \\ & (\exists j \in Adj_i : j \notin i[cliq] \wedge (\forall k \in i[cliq] : k \in Adj_j)). \end{aligned}$$

The algorithm that we develop next is a self-stabilizing algorithm, which means that the nodes can be initialized arbitrarily. Thus, $i[cliq]$ may contain the nodes that are not connected to i by an edge. The algorithm is defined as follows. If all the nodes in $i[cliq]$ do not form a clique, then $i[cliq]$ is reset to be $\{i\}$. If there exists some node j in Adj_i such that $i[cliq] \cup \{j\}$ is a clique, but j is not in $i[cliq]$, then j is added to $i[cliq]$.

Algorithm 6.1. *Rules for node i in state s .*

$$\boxed{\begin{array}{l} \text{IMPEDENSABLE-DC}(i) \longrightarrow \\ \left\{ \begin{array}{ll} i[\text{cliq}] = \{i\} & \text{if } (i \notin i[\text{cliq}] \vee (\exists j, k \in i[\text{cliq}] : j \neq k \wedge j \notin \text{Adj}_k)) \\ i[\text{cliq}] = i[\text{cliq}] \cup \{j\} & \text{otherwise} \end{array} \right. \\ \text{(where } j \text{ is such that } \forall i \in i[\text{cliq}] : j \in \text{Adj}_i) \end{array}}$$

Lemma 6.1. *The dominant Clique problem is a PO-inducing problem.*

Proof. For a node i , $i[\text{cliq}]$ contains the nodes that i is connected with, and the nodes in $i[\text{cliq}]$ should form a clique. A global state does not manifest a dominant clique if at least one node i in s does not store a set of nodes forming a maximal clique with itself, i.e. (1) $i[\text{cliq}]$ is not a maximal clique, that is, there exists a j in $\text{Adj}_i \setminus i[\text{cliq}]$ such that $i[\text{cliq}] \cup \{j\}$ forms a valid clique, or (2) the nodes in $i[\text{cliq}]$ do not form a clique.

Next, we need to show that if some node i in state s is violated, then for any global state s' such that $s' \succeq s$, if $s'[i] = s[i]$, then s' will not manifest a dominant clique. This is straightforward from the definition itself, that if a node i is impedensable, then i does not store a set of nodes forming a maximal clique with itself. Thus, if i is impedensable in s , and i has the same state in some s' such that $s' \succ s$, then s' , as well, does not satisfy \mathcal{P}_{dc} . \square

To present the abstraction of the partial order induced among the local states, we define the state value of a local state as follows.

$$\text{STATE-VALUE-DC}(i, s) = \begin{cases} |C| - |i[\text{cliq}]| : C = \text{largest superset of } i[\text{cliq}] \text{ that is a valid clique} & \text{if } i[\text{cliq}] \text{ is a clique.} \\ \text{deg}(i) + 1 & \text{otherwise.} \end{cases}$$

\mathcal{P} induces a partial order among the local states, which can be abstracted by state value as defined above: for a pair of global states s and s' , $s[i] \prec s'[i]$ iff $\text{STATE-VALUE-DC}(i, s') < \text{STATE-VALUE-DC}(i, s)$. As an instance, the partial order induced among the local states of node v_1 (of the graph in Figure 6.1 (a)) is shown in Figure 6.1 (b).

To present the abstraction of the \prec -DAG induced among the global states, we define the rank of a global state as follows.

$$\text{RANK-DC}(s) = \sum_{i \in V(G)} \text{STATE-VALUE-DC}(i, s).$$

Under Algorithm 6.1, the global states of the graph in Figure 6.1 (a) form a \prec -DAG that we show in Figure 6.2. For a pair of global states s and s' , $s \prec s'$ iff $\text{RANK-DC}(s') < \text{RANK-DC}(s)$. In Figure 6.2, a global state is represented as $\langle\langle v_1[\text{cliq}], v_2[\text{cliq}], v_3[\text{cliq}] \rangle\rangle$. The state space for this instance has a total

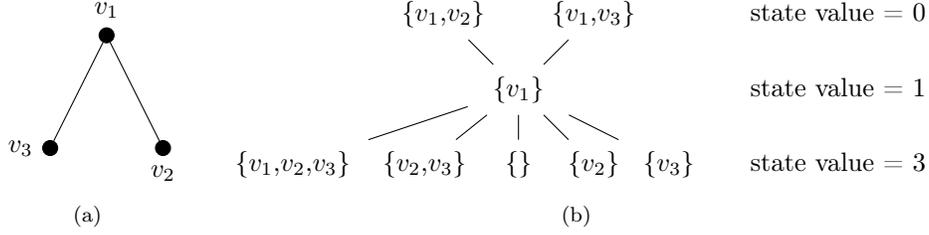


Figure 6.1: (a) Input graph. (b) The acronym s.v. stands for state value. Partial order among local states of node 1 (all edges are directed upwards).

512 states. In the figure, we only show the states where the second guard is false in all the nodes. All the global states where the second guard is true in some nodes will converge to one of the states present in this figure, and then it will converge to one of the terminal successors.

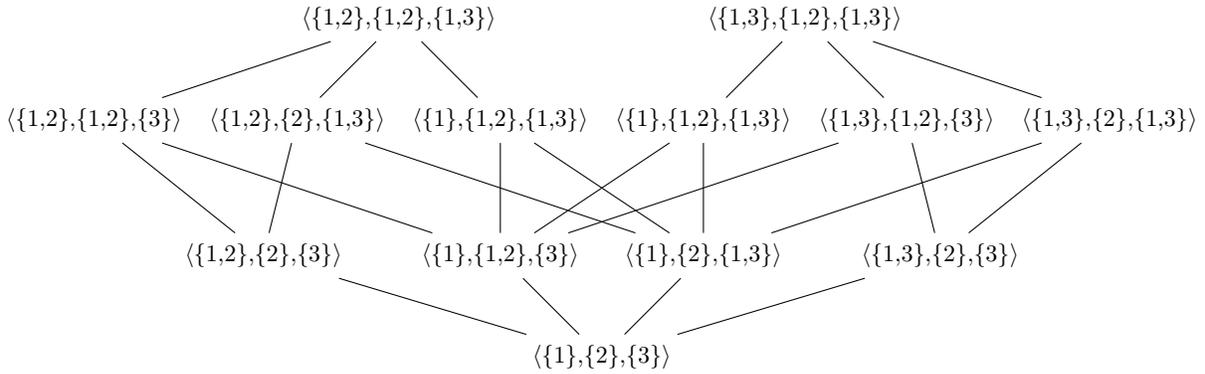


Figure 6.2: \leftarrow -DAG, assuming that initial state is $\langle\{1\}, \{2\}, \{3\}\rangle$; we replaced writing v_i by i for brevity. In all these states, the second guard of Algorithm 6.1 is false. Observe that any other state will converge to one of these states and then converge to one of the optimal states in this \leftarrow -DAG. (Transitive edges are not shown; all edges are directed upwards.)

Notice that $\forall i$ -IMPEDENSABLE-DC(i) is a self-stabilizing PO-inducing predicate and satisfies Definition 6.8; Algorithm 6.1 that utilizes this predicate is a self-stabilizing algorithm.

Theorem 6.1. *Algorithm 6.1 is a silent self-stabilizing algorithm for the dominant clique problem on n nodes executing asynchronously.*

Proof. We need to show that (1) Algorithm 6.1 traverses a \leftarrow -DAG of global states, (2) for all suboptimal states, \exists a terminal successor, and (3) all terminal global states are optimal states.

Let the current state be s . If s is suboptimal, then for at least for one of the nodes i : (1) $i[cliq]$ is not a maximal clique, that is, there exists a j in $Adj_i \setminus i[cliq]$ such that $i[cliq] \cup \{j\}$ forms a valid clique, or (2) the nodes in $i[cliq]$ do not form a clique.

In the case that s is suboptimal and the first case holds true for some node i , then under Algorithm 6.1,

i will include a node j in $i[cliq]$ which forms a clique with the nodes already present in $i[cliq]$, which reduces the state value of i by at least 1.

In the case that s is suboptimal and the second case holds true for some node i , then under Algorithm 6.1, i will change $i[cliq]$ to be $\{i\}$, which reduces the state value of i from $deg(i) + 1$ to some value less than or equal to $deg(i)$.

This shows a partial order being induced among the local states visited by an arbitrary node i . Thus under Algorithm 6.1, an arbitrary graph will follow a \prec -DAG of states and if it transitions from a state s to another state s' , then we have that $s' \succ s$ such that rank of s' is less than the rank of s .

If some node is impedensable, then the rank of the corresponding global state is non-zero. When a impedensable node i makes an execution, then its state value reduces, until it becomes 0. Thus if there is a global state s with rank greater than 0, then there exists at least one impedensable node in it. When any node performs execution in s then s transitions to some state with rank less than s . This shows that for every suboptimal global state, there exists at least one terminal successor.

Let that s is a terminal successor. This implies that $\mathcal{P}(s)$ is true: no node is impedensable in s , so any node will not change its state and s manifests a dominant clique. Thus we have that all terminal states are optimal states, and Algorithm 6.1 is silent. \square

Algorithm 6.1 is a distance-1 algorithm and guarantees converges in asynchrony. This is because in a given state s some node i is impedensable iff i does not store a dominant clique, thus, s will never transition to an optimal state without i changing its state.

6.1.4 Shortest Path (SP) Problem

Definition 6.10. Shortest path. *In the shortest path problem, the input is a weighted arbitrary connected graph G (all edge weights are positive) and a destination node v_{des} . Every node i stores $i[p]$ (initialized with \top) and $i[d]$ (initialized with ∞). The task is to compute, $\forall i \in V(G)$, the length $i[d]$ of a shortest path from i to v_{des} , and the parent $i[p]$ through which an entity would reach v_{des} starting from i .*

The positive weights assigned for every edge $\{i, j\} \in E(G)$ denote the cost that is required to move from node i to node j . In this problem, if we would have considered the local state of a node i to be represented only by the variable $i[d]$ then the local states of the nodes would form a total order. Consequently, the resultant discrete structure formed among the global states will be a \prec -lattice. This was shown in [6]. On the other hand, in applications such as source routing [18] where the source node specifies the path that should be taken, the local states form a partial order: such a system cannot be simulated within a total order. For brevity, we only represent the next hop, in $i[p]$. The SP problem can be represented by the following predicate, where, $w(i, j)$ is the weight of edge $\{i, j\}$.

$$\mathcal{P}_{sp} \equiv \forall i : (i[d] = \text{dis}(i, v_{des}) = \min\{\text{dis}(j, v_{des}) + w(i, j) : j \in \text{Adj}_i\}) \wedge \\ (i[p] = \arg \min\{\text{dis}(j, v_{des}) + w(i, j) : j \in \text{Adj}_i\}).$$

The local state of a node i is defined by $\langle i[p], i[d] \rangle$. An impedensable node i in a state s is a node for which its current parent is not a direct connection to the shortest path from i to v_{des} . Formally,

$$\text{IMPEDENSABLE-SP}(i) \equiv (i[d] \neq 0 \wedge i = v_{des}) \vee (\exists j \in \text{Adj}_i : i[d] > j[d] + w(i, j)).$$

The algorithm that we develop next is not a self-stabilizing algorithm. We require that each node i has $i[d]$ initialized to ∞ , however, $i[p]$ can be arbitrarily initialized. However, as we later prove in this subsection, this algorithm converges even without enforcing any synchronization mechanism. The algorithm is defined as follows. If an impedensable node i is v_{des} , then $i[d]$ is updated to 0 and $i[p]$ is updated to v_{des} . Otherwise, $i[p]$ is updated to the j in Adj_i for which $j[d] + w(i, j)$ is minimum.

Algorithm 6.2. *Rules for node i .*

| |
|---|
| $\text{IMPEDENSABLE-SP}(i) \longrightarrow \begin{cases} i[d] = 0, i[p] = i & \text{if } i = v_{des} \\ \langle i[d], i[p] \rangle = \langle j[d] + w(i, j), j \rangle : j = \arg \min\{k[d] + w(i, k) : k \in \text{Adj}_i\} & \text{otherwise} \end{cases}$ |
|---|

We show in the following that Algorithm 6.2 is a PO-inducing algorithm, and the properties of this algorithm imply that the SP problem is a PO-inducing problem.

Lemma 6.2. *The shortest path problem is a PO-inducing problem.*

Proof. For a node i , $i[d]$ contains the distance of v_{des} from node i . A global state s does not manifest all correct distances if for at least one node i in s , (1) $\text{dis}(i, v_{des}) \neq i[d]$, that is, i does not store a shortest path from i to v_{des} , or (2) the parent of i is not a valid direct connection in a shortest path from i to v_{des} .

Next, we need to show that if some node i in state s violates \mathcal{P}_{sp} , then for each global state s' such that $s' \succ s$, if $s'[i] = s[i]$, then s' will not manifest all shortest paths. This is straightforward from the definition itself, that if a node i is impedensable, then either $i = v_{des}$ and it is not pointing to itself through $i[p]$, or there is at least one other node j such that $i[d] > j[d] + w(i, j)$. If i is impedensable in s , and i has the same state in some global state s' such that $s' \succ s$, then i stays impedensable in s' as well, and s' does not satisfy \mathcal{P}_{sp} . □

To present the abstraction of the induction of an \prec -DAG, we define the state value and rank as follows.

$$\text{STATE-VALUE-SP}(i, s) = i[d] - \text{dis}(i, v_{des}). \\ \text{RANK-SP}(s) = \sum_{i \in V(G)} \text{STATE-VALUE-SP}(i, s).$$

Under Algorithm 6.2, the global states form a \prec -DAG. We show an example in Figure 6.3. Figure 6.3 (a) is the input graph and Figure 6.3 (b) is the \prec -DAG induced among the global states. For a pair of global states s and s' , $s \prec s'$ iff $\text{RANK-SP}(s') < \text{RANK-SP}(s)$. In Figure 6.3, a global state is represented as $\langle\langle v_1[p], v_1[d]\rangle, \dots, \langle v_4[p], v_4[d]\rangle\rangle$.

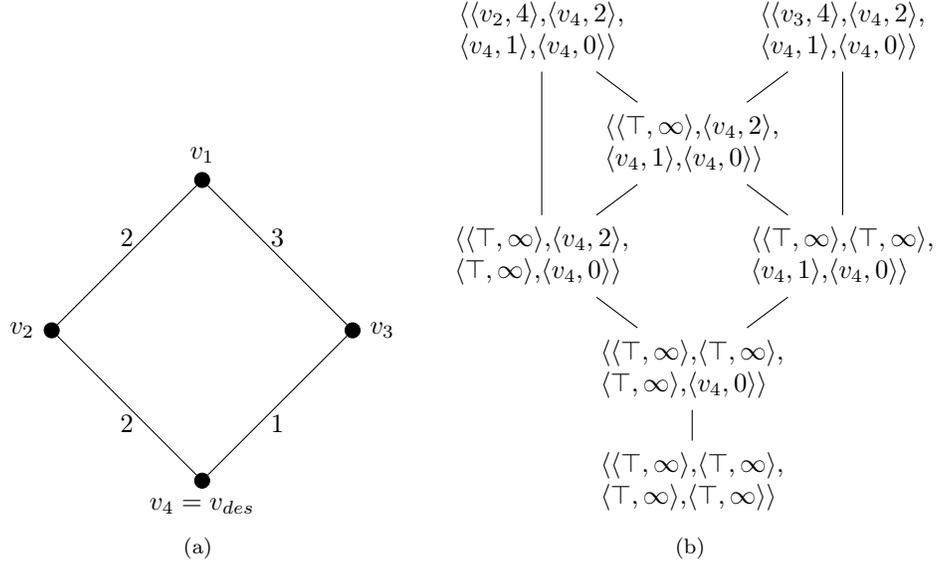


Figure 6.3: (a) Input graph. (b) \prec -DAG induced among the global states in evaluating for the shortest path problem in the graph shown in (a); a global state is represented as $\langle\langle p.v_1, d.v_1 \rangle, \dots, \langle p.v_4, d.v_4 \rangle\rangle$. Transitive edges are not shown.

The above algorithm requires that all nodes are initialized where $i[d] = \infty, i[p] = \top$. If nodes were initialized arbitrarily (e.g., if for all nodes i , $i[d] = 0$) then the algorithm does not compute shortest paths. Hence, $\forall i \neg \text{IMPEDENSABLE-SP}(i)$ is a PO-inducing predicate but is not self-stabilizing; in turn, Algorithm 6.2, that utilizes this predicate, is not self-stabilizing.

Theorem 6.2. *Algorithm 6.2 solves the shortest path problem, on a connected positive weighted graph, on n nodes executing asynchronously.*

Proof. We need to show that (1) Algorithm 6.2 traverses a \prec -DAG of global states, (2) for all suboptimal states, \exists a terminal successor, and (3) all terminal global states are optimal states.

Let the current state be s . If s is suboptimal, then for at least one of the nodes i : (1) $i[p] \neq i \wedge i = v_{des}$, that is, i is the destination node and is not pointing to itself, or (2) $\text{dis}(i, v_{des}) \neq i[d]$, that is, i does not store a shortest path from i to v_{des} .

In the case that s is suboptimal and the first case holds true for some node i , then under Algorithm 6.1, i updates $i[d]$ to 0 and $i[p]$ to i , which reduces the state value of i to 0.

In the case that s is suboptimal and the second case holds true for some node i , then under Algorithm 6.2, i will reduce its $i[d]$ value and update $i[p]$, which reduces the state value of i at least by 1.

This shows that a partial order is induced among the local states visited by an arbitrary node i . Thus under Algorithm 6.1, an arbitrary graph will follow a \prec -DAG of global states and if it transitions from a state s to another state s' , then we have that $s' \succ s$ such that rank of s' is less than the rank of s .

If no node is impedensable, then this implies that all nodes have computed the shortest distance in their $i[d]$ variable, and thus the rank is 0. Thus if there is a global state s with rank greater than 0, then there exists at least one impedensable node in it. When any node performs execution in s then s transitions to some state s' such that the rank of s' is less than the rank of s . This shows that for every suboptimal global state, there exists at least one terminal successor.

Let that s is a terminal successor. Then, $\mathcal{P}(s)$ is true: no node is impedensable in s , so any node will not execute and s manifests correct shortest path evaluation for all nodes. Thus we have that all terminal states are optimal states, and Algorithm 6.2 is silent. \square

Algorithm 6.2 is a distance-1 algorithm and converges even if the nodes run without synchronization in AA model. This is because in a given state s , some node i is impedensable iff there is a shorter path that i can follow to reach v_{des} , thus, s will never transition to an optimal state without i changing its state.

6.1.5 Limitations of Modelling Problems as PO-Inducing Problems

Unlike the PO-inducing problems where the problem description creates a \prec -DAG among the states in S , there are problems where the states do not form a \prec -DAG naturally. Such problems are non-PO-inducing problems. In such problems, there are instances in which the impedensable nodes cannot be distinctly determined, i.e., in those instances $\exists s : \neg \mathcal{P}(s) \wedge (\forall i : \exists s' : \mathcal{P}(s') \wedge s[i] = s'[i])$.

Definition 6.11. Maximal matching. *In the maximal matching problem, the input is an arbitrary graph G . For all i , $i[match]$ has the domain $Adj_i \cup \{\top\}$. The task is to compute the $i[match]$ (for each node i) such that (1) $\forall i : i[match] \neq \top \Rightarrow (i[match])[match] = i$, and (2) if $i[match] = \top$, then there must not exist a j in Adj_i such that $j[match] = \top$.*

Maximal matching (MM) is a non-PO-inducing problem. This is because, for any given node i , an optimal state can be reached if i does or does not change its state. Thus i cannot be deemed as impedensable or not impedensable under the natural constraints of MM. This can be illustrated through a simple instance of a 3 nodes network forming a simple path $\langle A, B, C \rangle$. Initially no node is paired with any other node. Here, MM can be obtained by matching A and B . Thus, C is not impedensable. Another maximal matching can be obtained by matching B and C , in which case A is not impedensable. Thus the problem itself does not define which node is impedensable.

We observe that it is possible to induce a \prec -DAG in non-PO-inducing problems algorithmically. We call such algorithms non-PO-inducing algorithms, which we study in the following section.

6.2 Imposed PO-Induction: PO-inducing Algorithms

In this section, we study algorithms that can be developed for problems that cannot be represented by a predicate under which the global states form a \prec -DAG. This is because, as described in Section 6.1.5, in a suboptimal global state, the problem does not specify a specific set of nodes that must change their state.

6.2.1 General Properties of PO-Inducing Algorithms

Non-PO-inducing problems do not naturally define which node is impedensable. There may be multiple optimal states. However, impedensable nodes can be defined algorithmically.

Definition 6.12. PO-inducing algorithms (DIA). *A is a DIA for a problem P , represented by predicate \mathcal{P} , iff (1) P is solved iff the system reaches a state where \mathcal{P} is true, and (2) \mathcal{P} is PO-inducing with respect to \mathcal{S} induced in \mathcal{S} by A , i.e. $\forall s \in \mathcal{S} : \neg \mathcal{P}(s) \Rightarrow \exists i : \text{IMPEDENSABLE}(i, s, \mathcal{P})$.*

Remark: An algorithm that traverses a \prec -DAG \mathcal{S} of global states is a DIA. Thus, an algorithm that solves a PO-inducing problem, under the constraints of PO-induction, e.g. Algorithm 6.1, is a DIA.

Remark: Since a total order is a special case of a partial order, all lattice-linear algorithms are PO-inducing algorithms.

Definition 6.13. Self-stabilizing DIA. *Continuing from Definition 6.12, A is self-stabilizing only if in the \prec -DAG \mathcal{S} induced by A , $\forall s, s' \in \mathcal{S} : \text{TERMINAL-SUCCESSOR}(s, s') \Rightarrow \mathcal{P}(s') = \text{true}$.*

In the remaining part of this section, we study the maximal matching problem, a non-PO-inducing problem, and explore how a PO-inducing algorithm can be developed for such a problem. We will see, again, that the induction of a partial order among the local states, under the acting algorithm, allows the nodes to execute without synchronization.

6.2.2 Maximal Matching (MM) Problem

As discussed in Section 6.1.5, MM is not a PO-inducing problem. However, a PO-inducing algorithm can be developed for this problem, which we discuss in the following.

The local state of a node i is defined by $\langle i[\text{match}] \rangle$. We use the macros listed in Table 6.1. A node i is *wrongly matched* if i is pointing to some node j , but j is pointing to some node $k \neq i$. A node i is *matchable* if i is not pointing to any node, i.e. $i[\text{match}] = \top$, and there exists a node j adjacent to i which is also not pointing to any node. A node i is being *pointed to*, or i is *i -pointed*, if i is not pointing to any node, and there exists a node j adjacent to i which is pointing to i . A node sees that another node is being pointed, or i “sees” *else-pointed*, if some node j around (in 2-hop neighbourhood of) i is pointing to another node

k and k is not pointing to anyone. A node is *unsatisfied* if it is wrongly matched or matchable. A node i is *impedensable* if i is i -pointed, or otherwise, given that i does not see else-pointed, i is the highest ID unsatisfied node in its distance-2 neighbourhood.

| |
|---|
| $\text{WRONGLY-MATCHED-MM}(i) \equiv i[\text{match}] \neq \top \wedge (i[\text{match}])[\text{match}] \neq i \wedge (i[\text{match}])[\text{match}] \neq \top.$ $\text{MATCHABLE-MM}(i) \equiv i[\text{match}] = \top \wedge (\exists j \in \text{Adj}_i : j[\text{match}] = \top).$ $\text{I-POINTED-MM}(i) \equiv i[\text{match}] = \top \wedge (\exists j \in \text{Adj}_i : j[\text{match}] = i).$ $\text{ELSE-POINTED-MM}(i) \equiv \exists j \in \text{Adj}_i^2, \exists k \in \text{Adj}_j : j[\text{match}] = k \wedge k[\text{match}] = \top.$ $\text{UNSATISFIED-MM}(i) \equiv \text{WRONGLY-MATCHED-MM}(i) \vee \text{MATCHABLE-MM}(i).$ $\text{IMPEDENSABLE-MM}(i) \equiv \text{I-POINTED-MM}(i) \vee (\neg \text{ELSE-POINTED-MM}(i) \wedge (\text{UNSATISFIED-MM}(i) \wedge (\forall j \in \text{Adj}_i^2 : i[id] > j[id] \vee \neg \text{UNSATISFIED-MM}(j)))).$ |
|---|

Table 6.1: Macros used in the algorithm for MM.

The algorithm that we develop next is a self-stabilizing algorithm, which means that the nodes can be initialized arbitrarily. Thus, $i[\text{match}]$ may store some node that is not connected to i by an edge, or some node j in Adj_i but $j[\text{match}]$ may not store i . The algorithm for an arbitrary node i can be defined as follows. If i is impedensable and i -pointed, then i starts to point to the node which is pointing at i . If i is wrongly matched and impedensable, then i takes back its pointer, i.e. i starts pointing to \top . Otherwise (if i is matchable and impedensable), i chooses a node j which is not pointing to anyone, i.e. $j[\text{match}] = \top$, and i starts pointing to j .

Algorithm 6.3. *Rules for node i .*

| |
|---|
| $\text{IMPEDENSABLE-MM}(i) \longrightarrow$ $\left\{ \begin{array}{ll} i[\text{match}] = j : j \in \text{Adj}_i : j[\text{match}] = i & \text{if I-POINTED-MM}(i). \\ i[\text{match}] = \top & \text{if WRONGLY-MATCHED-MM}(i). \\ i[\text{match}] = j : j \in \text{Adj}_i : j[\text{match}] = \top & \text{otherwise.} \end{array} \right.$ |
|---|

Lemma 6.3. *Algorithm 6.3 induces a \prec -DAG in the global state space under AMR model.*

Proof. The \prec -DAG is induced in the global state space with respect to the state values, which we prove in the following. Let s be a suboptimal state that the input graph is in. A node i is impedensable in s (1) if i is wrongly matched, (2) if i is matchable, or (3) if i is being pointed at by another node j , but i does not point back to j or any other node. We elaborate on all these cases in the following paragraphs of this proof. In all these cases, we assume AMR model, that is, if a node reads a local state of another node, then it does not read an older local state from that node in a subsequent read operation.

We show that if some node i is impedensable in some state s , then for any state $s' : s' \succ s$, if $s'[i] = s[i]$, then s' will not form a maximal matching under Algorithm 6.3.

In the case if i is wrongly matched in s and is impedensable, and it is pointing to the same node in s' as well, then i stays to be impedensable in s' . This is because since i is impedensable, i is also the highest ID node that is unsatisfied, so all other nodes within distance-2 of i will wait for i to take back its pointer before taking any action. Thus s' does not have a correct matching.

In the case if in s , i is being pointed to by some node but i does not point to any node, and i stays in the same state in s' , then i stays to be impedensable in s' . This can be explained as follows. Let j be the node that is pointing to i , i.e., $j[match] = i \wedge i[match] = \top$. Now since $\text{I-POINTED-MM}(i)$ is true, so for all unsatisfied nodes within distance-2 of i , ELSE-POINTED is true. Thus, they will not take any action until i does. Also since $i[match] = \top$, j will not retreat its pointer as it does not fall under the constraints of WRONGLY-MATCHED . Thus s' does not form a correct matching.

Finally, in the case if i is matchable and impedensable in s' , and it stays the same in s' , then it is still impedensable as any other node in Adj_i will not initiate matching with it. This is because all the unsatisfied nodes within distance-2 of i have IDs less than that of i . Also, for the same reason, any node in Adj_i^2 will not take any action until i does. Thus s' does not manifest a maximal matching. \square

To present the abstraction of the induction of an \prec -DAG under Algorithm 6.3, we define the state value and rank as follows.

$$\text{STATE-VALUE-MM}(i, s) = \begin{cases} 3 & \text{if } \text{WRONGLY-MATCHED-MM}(i). \\ 2 & \text{if } \text{MATCHABLE-MM}(i) \wedge \neg \text{I-POINTED-MM}(i). \\ 1 & \text{if } \text{I-POINTED-MM}(i). \\ 0 & \text{otherwise.} \end{cases}$$

$$\text{RANK-MM}(s) = \sum_{i \in V(G)} \text{STATE-VALUE-MM}(i, s).$$

Under Algorithm 6.3, the global states form a \prec -DAG. We show an example in Figure 6.4: Figure 6.4 (a) is the input graph and Figure 6.4 (b) is the induced \prec -DAG. For a pair of global states s and s' , $s \prec s'$ iff $\text{RANK-MM}(s') < \text{RANK-MM}(s)$. In Figure 6.4, a global state is represented as $\langle \langle v_1[match] \rangle, \dots, \langle v_4[match] \rangle \rangle$.

Since the solution presented for this problem is self-stabilizing, $\forall i \neg \text{IMPEDENSABLE-MM}(i)$ forms a self-stabilizing predicate with respect to the \prec -DAG induced by Algorithm 6.3. Thus, Algorithm 6.3 is a PO-inducing self-stabilizing algorithm and satisfies Definition 6.13.

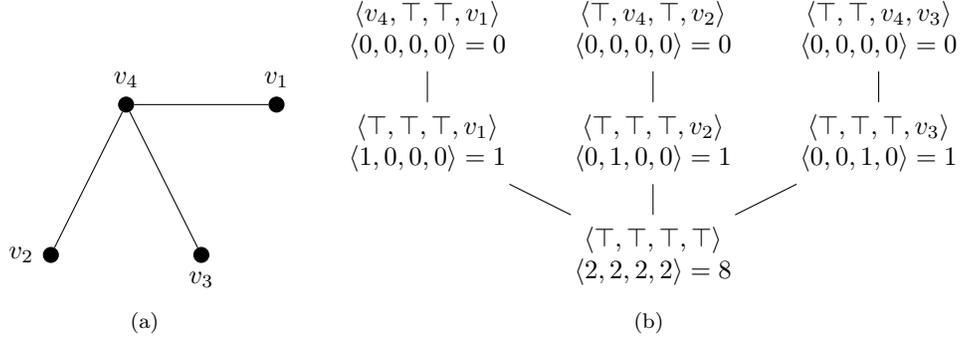


Figure 6.4: (a) Input graph. (b) The state transition diagram, a \prec -DAG, assuming that the initial global state is $\langle \top, \top, \top, \top \rangle$. In every state, the first row shows the global state, the second row shows the respective local state values of nodes and the rank of the global state. Observe that any other state will converge to one of these states and then converge to one of the optimal states in this \prec -DAG. (All edges are directed upwards; transitive edges are not shown for brevity).

Theorem 6.3. *Algorithm 6.3 is a PO-inducing algorithm for the maximal matching problem on n nodes executing asynchronously in AMR model.*

Proof of Theorem 6.3. We show that (1) Algorithm 6.3 traverses a \prec -DAG of global states, under AMR model, that has the properties as mentioned in the above lemma, (2) for all suboptimal states \exists a terminal successor, and (3) all terminal global states are optimal states.

If s is suboptimal, then for at least one of the nodes i : (1) i is wrongly matched, (2) i is matchable, or (3) i is being pointed at by another node j , but i does not point back to j or any other node.

If s is suboptimal and some node i is being pointed to by some node j and i does not point to any node, then under Algorithm 6.3, i will point back to j , and thus the state value of i will get reduced from 1 to 0.

In the case that s is suboptimal and some node j is wrongly matched or matchable with \neg ELSE-POINTED-MM(j), then at least one node (e.g., a node with highest ID which is wrongly matched or matchable) will be unsatisfied and impedensable. Let that i is unsatisfied and impedensable. Here i is either wrongly matched or matchable. If i is wrongly matched, then i will change its pointer and start pointing to \top , in which case its state value will change from 3 to 2, 1, or 0. If i is matchable then, i will start pointing to some node j in Adj_i , in which case, the state value of i will change from 2 to 0 and the state value of j will change from 2 to 1.

In all the above cases, we have that under Algorithm 6.3, if s is a suboptimal state, then its rank will be of some value greater than zero because at least one of the nodes will be impedensable. s will transition to some state s' whose rank is less than that of s . Thus, we have that Algorithm 6.3 transitions s to s' and thus decreases the rank of the system. This shows that (1) Algorithm 6.3 traverses a \prec -DAG that has the properties as mentioned in Lemma 6.3, (2) for all suboptimal states \exists a terminal successor.

In the case that s is a terminal successor, then none of the nodes will be enabled. So no node will change its state. This state will manifest a maximal matching. This shows that all terminal successors are optimal states, and Algorithm 6.3 is silent. \square

Algorithm 6.3 is a distance-4 algorithm. From Theorem 6.3, we have that Algorithm 6.3 converges even if the nodes run without synchronization in AMR model. However, Algorithm 6.3 cannot tolerate asynchrony in AA model. This is because a node i in its current state may wrongly evaluate itself to be i -pointed or unsatisfied-impedensable if it gets information, from other nodes, out of order. This can result in i changing its state incorrectly. As a consequence of execution in AA model, i can keep repeating such execution and as a result, the system may not obtain an optimal state.

6.3 Properties of PO-Induction

In the previous sections, we discussed example algorithms that converge even without synchronization. In this section, we use that intuition and describe some characteristics of general PO-inducing systems. We study how the induction of a partial order among the local states visited by individual nodes is a necessary and sufficient condition to allow asynchrony (Section 6.3.1). We also study some time-complexity properties of an algorithm that induces a \prec -DAG among the global states (Section 6.3.2).

6.3.1 PO-Induction to Obtain Asynchrony

In this subsection, we study whether \prec -DAG induction is necessary and sufficient for asynchronous execution.

Theorem 6.4. *Let P be a problem that requires an algorithm to converge to a state where \mathcal{P} is true. Let A be an algorithm for P that is correct under a central scheduler. Let \mathcal{S} be the transition graph that is formed under A , where nodes are allowed to read old values in some communication model M .*

If \mathcal{S} forms a \prec -DAG, then A guarantees convergence in asynchrony in the model M .

Proof. Definition 6.12 follows that the local states form a partial order if every node, under Algorithm A , rejects each violating local state permanently. Consequently, a \prec -DAG is induced among the global states.

A \prec -DAG, induced under \mathcal{P} , allows asynchrony because if a node, reading old values, reads the current state s as s' , then $s' \prec s$. So $\neg\mathcal{P}(s') \Rightarrow \neg\mathcal{P}(s)$ because $\text{IMPEDENSABLE}(i, s', \mathcal{P})$ and $s'[i] = s[i]$. \square

The limitation of the above theorem is as follows. An algorithm A guarantees convergence in asynchrony in some model only if it induces a \prec -DAG in that model. For instance, a Algorithm 6.3 developed for maximal matching induces a \prec -DAG in AMR model. As we discussed in Section 6.2.2, such an algorithm does not necessarily induce a \prec -DAG in AA model, so it may not guarantee convergence in AA model.

In the above theorem, we showed that a \prec -DAG is sufficient for allowing asynchronous executions. Next, we study if a \prec -DAG is guaranteed to be induced among the global states given that an algorithm is correct under asynchrony. In other words, we study if a \prec -DAG is necessary for asynchrony. To study that, we first examine the representation of a global state, which is a mathematical abstraction of a multiprocessor system.

Let C be an algorithm that runs correctly under a central scheduler. Let that R_s^{cent} be the set of states, that s can transition to, under C , i.e. $\forall s' \in R_s^{cent}, \langle s, s' \rangle$ is a valid transition under C . If C and s are given, R_s^{cent} can be correctly computed. In synchronous systems, s is an abstraction of the global state that only contains the *current* local states of the nodes. However, if some algorithm A were to be executed in asynchrony, then a set R_s^{asyn} of resulting states cannot be computed correctly for a given state s . This is because some nodes would be reading the old values of other nodes. Thus, s must also contain the details of the information that nodes have about other nodes.

Let \mathcal{S} be the original transition system, and \mathcal{S}_{ext} be its extended version, where a given state s_{ext} identifies local states of individual nodes and the information that the nodes maintain about other nodes. Similarly, \mathcal{S}_{ext} can be constructed back to \mathcal{S} , by removing the information that the nodes have about other nodes, and then merging the resulting global states (along with the transition edges) that are the same.

Observe that in all the algorithms presented in this chapter, the STATE-VALUE of a node i provides information about how *bad* the current local state of i is, with respect to an optimal global state farthest from its current global state s . Thus, the evaluation of the state value of i can utilize the information about the local states of other nodes in s . In the following theorem, we use this observation as leverage to show some interesting properties of PO-inducing systems. Herein, instead of \mathcal{S} we elaborate on the necessity of \mathcal{S}_{ext} being a \prec -DAG to allow asynchrony.

Theorem 6.5. *Let P be a problem that requires an algorithm to converge to a state where \mathcal{P} is true. Let A be an algorithm for P that is correct under a central scheduler. Let \mathcal{S} be the transition graph that is formed under A , where nodes are allowed to execute asynchronously in some communication model M .*

If algorithm A guarantees convergence in asynchrony under the communication model M , then \mathcal{S}_{ext} (the extended transition system) forms a \prec -DAG in the model M .

Proof. Since A guarantees to terminate under asynchrony and the extended state space \mathcal{S}_{ext} captures the effect of asynchrony (where a node may read old values of the variables of other nodes), there will be no cycles present among the global states in \mathcal{S}_{ext} , i.e., \mathcal{S}_{ext} forms a DAG. Next, we transform \mathcal{S}_{ext} to a \prec -DAG.

Let S_o^{ext} be the set of optimal states. For each optimal global state $o \in S_o^{ext}$, for each node i in o , assign

the state value of i to be 0. Thus, the rank of o is 0 (sum of state values of all nodes). For every non-optimal global state s , we assign the state value of every node i to be \top . Subsequently, if all successors of state s have a non-null (non- \top) rank (i.e., $\forall s' : \langle s, s' \rangle \in E(\mathcal{S}_{ext}) \Rightarrow (\forall i : \text{STATE-VALUE}(s'[i]) \neq \top)$) then we set the state value of each node i in s to be $(\max\{\text{RANK}(s') | \langle s, s' \rangle \in E(\mathcal{S}_{ext})\})/n + 1$. We do this recursively until no more updates happen to any state value of any global state in \mathcal{S}_{ext} .

Since there are no cycles, this procedure will terminate in finite time. Observe that using the above procedure, we obtain a valid \prec -DAG from \mathcal{S}_{ext} : in this \prec -DAG, for all reachable global states s' and s'' , $s' \prec s''$ (i.e., $\langle s', s'' \rangle$ is in $E(\mathcal{S}_{ext})$) iff $s'[i] \preceq s''[i], \forall i : [1 : n]$. \square

If \mathcal{S}_{ext} forms a \prec -DAG then so does \mathcal{S} , Hence, from Theorem 6.4 and Theorem 6.5, we have

Corollary 6.1. *Let P be a problem that requires an algorithm to converge to a state where \mathcal{P} is true. An Algorithm A guarantees convergence in asynchrony iff it induces a \prec -DAG among the (extended) global states.*

The above corollary shows that a \prec -DAG is a necessary and sufficient condition for a parallel processing system to guarantee convergence in asynchrony. For reasons of space, we have moved the subsection on the time complexity properties of PO-inducing algorithms to Appendix 6.3.2.

6.3.2 Time Complexity Properties of an Algorithm Traversing a \prec -DAG

Theorem 6.6. *Given a system of n processes, with the domain of (state values having) size not more than m for each process, the acting algorithm will converge in $n \times (m - 1)$ moves.*

Proof. Assume for contradiction that the underlying algorithm converges in $x \geq n \times (m - 1) + 1$ moves. This implies, by pigeonhole principle, that at least one of the nodes i is revisiting their state st after changing to st' . If st to st' is a step ahead transition for i , then st' to st is a step back transition for i and vice versa. For a system where the global states form a \prec -DAG, we obtain a contradiction since step-back actions are absent in such systems. \square

Example 6.MDS: continuation 4. *Consider phase 2 of Algorithm 5.1. As discussed earlier, this phase is lattice-linear. The domain of each process $\{IN, OUT\}$ is of size 2. Hence, phase 2 of Algorithm 5.1 requires at most $n \times (2 - 1) = n$ moves. (Phase 1 also requires atmost n moves. But this fact is not relevant with respect to Theorem 6.6.)* \square

Example 6.SMP: continuation 3. *Observe from Figure 2.2 that any system of 3 men and 3 women with arbitrary preference lists will converge in $3 \times (3 - 1) = 6$ moves. This comes from 3 men (resulting in 3 processes) and 3 women (domain size of each man (process) is 3).* \square

Corollary 6.2. *Let that each node i stores at most r variables, $i[\text{var}_1], \dots, i[\text{var}_r]$ (with domain sizes z_1, \dots, z_r respectively) contribute independently to the formation of the partial order. Then an algorithm traversing the resultant \prec -DAG will converge in $n \times ((\prod_{j=1}^r z_j) - 1)$ moves.*

Corollaries from Chapter 5

Corollary 6.3. *(From Theorem 5.1 and Theorem 6.6) Algorithm 5.2 converges in n moves.*

Corollary 6.4. *(From Theorem 5.2 and Theorem 6.6) Algorithm 5.3 converges in $\sum_{i \in V(G)} \text{deg}(i) + 1 = n + 2m$ moves.*

Proof. This can be reasoned as follows: first, a node may increase its colour, once, to resolve its colour conflict with a neighbouring node. Then it will decrease its colour, whenever it moves. Depending on the colour value of its neighbours and when they decide to move, a node i can decrease its colour almost $\text{deg}(i)$ times. □

Corollary 6.5. *(From Theorem 5.3 and Theorem 6.6) Algorithm 5.4 converges in n moves.*

Corollary 6.6. *(From Theorem 5.4 and Theorem 6.6) Algorithm 5.5 converges in n moves.*

Corollary 6.7. *(From Theorem 5.5 and Corollary 6.2) Algorithm 5.8 converges in n moves.*

Corollaries from this chapter

Corollary 6.8. *(From Theorem 6.1 and Corollary 6.2) Algorithm 6.1 converges in $\sum_{i \in V(G)} \text{deg}(i) = 2m$ moves. In terms of rounds, it converges in Δ rounds, where Δ is the maximum degree of the input graph.*

Corollary 6.9. *(From Theorem 6.2 and Corollary 6.2) Algorithm 6.2 converges in \mathcal{D} rounds, where \mathcal{D} is the diameter of the input graph.*

Corollary 6.10. *(From Theorem 6.3 and Corollary 6.2) Algorithm 6.3 converges in $2n$ moves.*

Proof. This is because, as explained in Lemma 6.3, any node i goes from state value 3 to 2 or 3 to 1, and then 2 to 0 or 1 to 0. Hence, there are at most two transitions that i goes through, with respect to its state value, which can happen due to the movement of i or some node in Adj_i . □

6.4 Summary of the Chapter

In this chapter, we focused on the problem of finding necessary and sufficient conditions for an algorithm to execute correctly without synchronization. We observe that the induction of a partial order among the local states is necessary and sufficient for multiprocessor algorithms to allow execution without any synchronization.

In PO-inducing problems, all unsatisfied nodes are enabled and can (and must) therefore evaluate their guards and take a corresponding action at any time. In a non-PO-inducing problem, however, all unsatisfied nodes are not enabled. Only the impedensable nodes are enabled; these nodes satisfy some additional constraints, in addition to being unsatisfied. In the algorithm for maximal matching, for example, we note that a tie-breaker is the key to deciding which nodes are impedensable; in algorithms for dominant clique and shortest path, on the other hand, do not require any tie-breaking strategy.

The sufficiency nature of \prec -DAG implies that such algorithms can be executed asynchronously, thereby eliminating the cost of synchronization. This is especially important in today's multiprocessor architecture where synchronization overhead is the Achilles heel of parallel algorithms. The necessity of this result means that an \prec -DAG exists in these algorithms even if the algorithms were designed without any prior assumption of an \prec -DAG.

Finally, we have that since a total order is a special case of a partial order, all lattice-linear systems are PO-inducing systems.

CHAPTER 7

RELATED WORK

In this chapter, we discuss works from several areas of computer science that we find related to our work.

7.1 Asynchronous Circuits

In this dissertation, we studied algorithms that are tolerant to asynchrony. There has also been recent development of hardware architectures that allow programs to run asynchronously, i.e., such architectures run programs without synchronization among their components. *Asynchronous circuits* are the circuits that do not synchronize their components centrally. Such circuits are also called *clockless* or *self-timed circuits*. We discuss some literature in this area, however, we do not go into the details of the hardware architecture of such circuits. The reader is directed to [19] for a comprehensive discussion on asynchronous circuits.

Asynchronous circuits contain communicating components, and the input and output ports of these components control the computation. On the other hand, in synchronous circuits, the computation is controlled by a global clock, that triggers the transition of the circuit from one state to another. Authors of [20] opine that asynchronous logic is the key technology for telecommunication applications.

Asynchronous circuits, as compared to synchronous circuits, have (1) low power consumption, (2) high performance, and (3) low noise and electromagnetic emissions [21]. This is mainly because, respectively, (1) asynchronous circuits are clock-driven whereas asynchronous circuits are data-driven, (2) asynchronous circuits are self-timed, and (3) asynchronous circuits implement a distributed control which results in low current peaks, as compared to synchronous circuits which implement a central control [20]. In addition, with the increase of the number and circuit size of the components on a chip, it is an increasingly complex problem to time all the components using a global clock, however, this problem is eliminated in asynchronous circuits.

The University of Manchester designed the AMULET2e, an embedded chip that incorporates a 32-bit ARM-compatible asynchronous core, a cache, and several other system functions [22–24]. The 80C51 microcontroller Philips Semiconductors and Philips Research, and then later redesigned to its asynchronous version [25]. 80C51, along with its successors – other integrated chips designed based on 80C51 – is the first asynchronous integrated chip that was commercially available. Cogency designed the Digital Signal Processor, and then later redesigned to its asynchronous version [26]. University of Osaka, University of Kochi and Sharp Corporation designed a self-timed data-driven multimedia processor [27–29]. An application for which it can be used is digital television receivers. Its peak performance is 8600 memory operations per second, and where it consumes below 1 watt.

Some other asynchronous chips are MiniMIPS designed by Caltech [30], AMULET3i designed by Uni-

versity of Manchester [31], TITAC2 designed by Tokyo University [32], and MICA designed by TIMA Laboratory [33]. The details of these chips are summarized in [20].

Asynchronous circuits have not received much attention in industry and academia. One of the contributing factors is the unavailability of software tools that can run on these circuits – programs that can tolerate asynchrony. This dissertation studies the necessary and sufficient conditions that make an algorithm tolerant to asynchrony. In addition, we not only develop new algorithms but also show that many existing algorithms in the literature are tolerant to asynchrony. This dissertation lays a theoretical foundation for asynchronous algorithms – that can run on asynchronous circuits – and provides ways to determine if an algorithm being developed is tolerant to asynchrony.

7.2 Other Abstractions in Concurrent Computing

In this section, we discuss some existing models that guarantee the progress or convergence of multiprocessor algorithms in the presence of node failures or the absence of synchronization at different levels.

7.2.1 Lock-Free and Wait-Free Algorithms

An algorithm is *non-blocking* if in a system running such algorithm, if a node fails or is suspended, then it does not result in failure or suspension of another node. A non-blocking algorithm is *lock-free* if system-wide progress can be guaranteed, and it is *wait-free* if progress can be guaranteed per node. A lock-free algorithm completes a given operation in a finite number of system steps, whereas a wait-free algorithm completes a given operation in a finite number of its own steps.

Non-blocking algorithms are very useful in designing memory transaction and input-output protocols due to the fact that such algorithms guarantee global (system-wide) or local (with respect to one computational node) progress. They allow processes that fail, in performing an operation due to contention, to continue processing other tasks and not continue to wait.

There is a vast literature on non-blocking algorithms. We note some of them as follows.

- Authors of [34] present algorithms for dynamic lock-free hash tables and list-based sets.
- A lock-free stack algorithm is presented in [35].
- Algorithms for implementing lock-free singly-linked lists are presented in [36].
- A lock-free algorithm to implement a binary search tree is present in [37].
- A wait-free sorting algorithm is studied in [38], which sorts an array of size N using $n \leq N$ processors.
- An $O(n)$ time wait-free approximate agreement algorithm is presented in [39].

A large class of lock-free algorithms, as shown in [40], under the scheduling conditions that are close to those implemented in commercially available hardware, stochastically behave as wait-free algorithms.

In the context of non-blocking algorithms, *contention* is the race condition that arises when multiple processes try to access the same resource (e.g., a variable, an array index or a memory location) simultaneously.

Lock-free algorithms are fast when contention is low, however, they do not provide an upper bound on the time complexity of individual operations when contention is high [41] (this paper uses the term ‘non-blocking’ to refer to lock-free algorithms). Authors of [42] showed that an adversary can cause $O(n)$ contention in a wait-free algorithm that is being executed by n processes in asynchrony.

There are some subtle differences between non-blocking and asynchronous algorithms. Non-blocking algorithms allow the nodes to return without waiting for an operation to complete, whereas asynchronous algorithms allow multiple nodes to perform operations concurrently such that the computing nodes do not block the progress of each other. Due possibility of contention, the scheduler is required to continuously check for the failed processes and completed tasks so that it can assign tasks to idle processes and guarantee progress, whereas asynchronous algorithms, the class of algorithms that this dissertation studies, eliminate the requirement to schedule tasks to processes.

In this dissertation, we are interested in asynchronous algorithms. Asynchronous algorithms are non-blocking, but not vice-versa. In addition, we do not assume a scenario where a node fails or turns byzantine. We assume that all nodes run correctly, however, their speeds can differ.

A key characteristic of PO-inducing algorithms is that they permit the algorithm to execute asynchronously. And, a key difference between non-blocking and asynchronous algorithms is the *system-perspective* for which they are designed. To understand this, observe that from a perspective, the asynchronous algorithms considered in this dissertation are wait-free. Each node reads the values of other nodes. Then, it executes an action, if it is enabled, without synchronization. More generally, in an asynchronous algorithm, each node reads the state of its relevant neighbours to check if the guard evaluates to true. It can, then, update its state without coordination with other nodes.

That said, the goal of asynchronous algorithms is not the progress / blocking of individual nodes (e.g., success of insert request in a linked list and a binary search tree, respectively, in [36] and [37]). Rather it focuses on the progress from the perspective of the system, i.e., the goal is not about the progress of an action by a node but rather that of the entire system. For example, in the algorithm for minimal dominating set present in this dissertation, if one of the nodes is slow or does not move, the system will not converge. However, the nodes can run without any coordination and they can execute on old values, instead of requiring a synchronization primitive to ensure convergence. In fact, the notion of impedensable (recall that in the algorithms that we present in this dissertation, in any global state, all enabled nodes are impedensable) captures this. An impedensable node has to make progress in order for the system to make progress.

7.2.2 Starvation-Free Algorithms

Starvation happens when requests of a higher priority prevent a request of lower priority from entering the critical section indefinitely. To prevent starvation, algorithms are designed such that the priority of

pending requests are increased dynamically. Consequently, a low-priority request eventually obtains the highest priority. Such algorithms are called *starvation-free* algorithms.

Starvation-free algorithms have been developed to solve several problems in multiprocessor systems. Some of the interesting works that deploy the starvation-free protocols are listed as follows.

- Authors of [43] present a starvation-free algorithm to schedule queued traffic in a network switch. This algorithm is based on the dynamic priority increment of the waiting requests.
- Authors of [44] implement a starvation-free distributed directory algorithm for shared objects. They show that this algorithm can serve concurrent requests and works correctly even in asynchrony.
- Authors of [45] introduce the notions of buffered semaphore and polite semaphore that facilitate starvation-free mutual exclusion. They showed, for three existing algorithms (present in, respectively, [32, 46, 47]), that they are implementations of one abstract algorithm and operate on one or the other of these semaphores.
- Authors of [48], on the other hand, view priority increment as a priority violation. They present an algorithm, for mutual exclusion in distributed systems, that postpones the priority increment of pending requests, and consequently the number of priority violations. Their algorithm is an extension of Karnar-Chaki algorithm [49]. They show that their methods (1) have a low message overhead (in comparison to Kanrar–Chaki algorithm and Chang’s priority-based algorithm [50]), (2) keep the same waiting time, and (3) tolerate the peaks of request load well.

In asynchronous algorithms, priority modification is not the key. For example, in the algorithms that we presented in this dissertation, we do not modify priority so as to ensure that some nodes can execute. Rather, the algorithms that are designed under the asynchronous model have the property that the nodes can execute independently given that there is at least one guard that holds true.

7.2.3 Serializability

Serializability in a distributed system allows only those executions to be executed concurrently which can be modelled to some permutation of a sequence of those executions. In other words, serializability does not allow nodes to read and execute on old information of each other: only those executions are allowed in concurrency such that reading fresh information, as if the nodes were executing in an interleaving fashion, would give the same result. Serializability is heavily utilized in database systems, and thus, the executions performed in such systems are called *transactions*.

- A *multidatabase system* is a system of multiple autonomous and heterogeneous (local) databases. Authors of [51] addressed the problem of how global serializability can be ensured in such a system. In their model, if some local database commits a pair of global transactions \mathcal{T}_1 and \mathcal{T}_2 in that sequence, but the local serialization is reversed, then such a schedule is aborted.

- Authors of [52] consider the problem in which the sequence of operations performed by a transaction may be repeated infinitely often. They describe a synchronization algorithm allowing only those schedules that are serializable in the order of commitment.
- Authors of [53] show that corresponding to several transactions, determining whether a sequence of read and write operations is serializable is an NP-Complete problem. They also present some polynomial time algorithms that approximate such serializability.
- JavaSpecs is a distributed data management tool produced by Sun Microsystem. Authors of [54] showed that serializability is satisfied in JavaSpaces only if we restrict to output, input, and read operations. Serializability, on the other hand, is not satisfied in the presence of test for absence or event notification.

In serializability protocols, schedules that cannot be serialized are aborted. However, in asynchronous systems, no process is aborted: all processes freely read from each other and perform executions independently. The asynchronous execution considered in this dissertation is not *serializable*, especially, since the reads can be from an old global state. Even so, the algorithm converges, and does not suffer from the overhead of synchronization required for serializability.

7.2.4 RedBlue Systems

In *redblue* systems (e.g., [55]), the rules can be divided into two non-empty sets: red rules, which must be synchronized, and blue rules, which can run in a lazy manner and do not have to be synchronized. Lattice-linear and asynchronous systems in general are the systems in which red rules are absent as an enabled node can execute independently regardless of which action is to be executed.

7.2.5 Local Mutual Exclusion

In *local mutual exclusion*, at a given time, some nodes block other nodes while entering to critical section. This can be done, e.g., by deploying semaphores.

- Authors of [56] study the group mutual exclusion problem, where nodes request for various “sessions” repeatedly, and it is required that (1) individual processes cannot be in different sessions concurrently, (2) multiple processes can be in the same session concurrently, and (3) if a process tries to enter a session, it is eventually able to do so.
- Authors of [57] propose a leader-based algorithm that deploys local mutual exclusion to solve resource allocation problem in Flying Ad hoc Networks.
- Authors of [58] presented an algorithm for distributed mutual exclusion in computer networks, that uses a spanning tree of the subject network. In this algorithm, the number of messages exchanged per critical section depends on the topology of this tree, typically this value is $O(n)$.
- Authors of [59] an algorithm with $O(\lg n)$ time complexity for mutual exclusion among n nodes. Specif-

ically, this algorithm requires atomic reads and writes and in which all spins are local (here a spin means a busy wait in which a node, in this case, waits on locally accessible shared variables).

We see, in algorithms based on local mutual exclusion, that they require additional data structures/variables to ensure that access is provided to (and blocking is deployed on) a certain set of processes. In asynchronous algorithms, nodes do not block each other. In non-lattice-linear problems, we see that usually a tie-breaker is required to ensure the correctness of the executions, however, if a problem is naturally lattice-linear, then it is not required. This is because in the case of non-lattice-linear problems, it may be desired that all unsatisfied nodes do not become enabled, however, in the case of lattice-linear problems, as we see in [6], all unsatisfied nodes can be enabled. And, all enabled nodes can read values and perform executions asynchronously, where they are allowed to read old values, which is not allowed in algorithms that deploy mutual exclusion.

7.3 Fixed Point Theorem

Fixed point theorems are extensively studied concepts in Mathematics. We discuss some closely related results from fixed point theory in this section, along with their applications. We include this discussion here because our work can be seen as a application of the fixed point theory – in our work, we develop systems, and theory of such systems, which converge at a global state, meaning that once an optimal global state is reached, then the system continues to be in that state for the rest of the execution. This is precisely the definition of a fixed point; we discuss the formal definition of this term in the following.

Let f be a function with the same domain and codomain. If for some input x , $f(x) = x$, then x is a *fixed point* of f . A function can have multiple fixed points. For example, let f_{ds} be a function that realizes the functionality of Algorithm 5.2 for the minimum dominating set: given a graph G , for an input global state s , f_{ds} will return the set of global states that G will transition to under Algorithm 5.2, if G is initialized in s . Observe, for example, in Figure 5.2, that all the supremum of each lattice is a fixed point for f_{ds} . In general, an optimal state, a state in which G manifests a minimal dominating set, is a fixed point for f_{ds} . Similarly, a function f_{smp} can be simulated for the algorithm for the stable marriage problem as presented in Example 2.3. It can be observed that for the instance of this problem as presented in Example SMP continuation 2 in Section 2.3, there are multiple global states that act as a fixed point for f_{smp} (a set of all states for this instance is shown in Figure 2.2). However, starting from the infimum of the lattice, i.e., $\langle 1, 1, 1 \rangle$ we only reach the state $\langle 1, 2, 2 \rangle$, which is one of the fixed points for this instance of the stable marriage problem. Note that all algorithms studied in this dissertation can be simulated, each, as a function with the same domain and codomain.

A *complete lattice* is a lattice for which there exists a unique infimum and a unique supremum. The lattices present in Figure 5.2 are, each, a complete lattice. The lattice present in Figure 2.2 is also a complete lattice.

A fixed-point theorem provides the conditions under which a fixed point exists in a system. We only study the fixed-point theorems that are most relevant to this dissertation.

There are some works that study fixed-point theorems in discrete systems, whereas some other works study fixed-point theorem in continuous systems.

It was shown in [60] that a function f whose domain and codomain are subsets of a set, which is increasing under set-theoretical inclusion, has at least one fixed point. This result was generalized in [61], which provides fixed point theorems in lattices. We discuss how the results in [61] relate to our work, next.

The lattices that we study in this dissertation are induced under the ‘ \preceq ’ operation. Let (L, \preceq) be a lattice, and let f be an order-preserving function with respect to the \preceq operation (e.g., f_{ds} and f_{smp}) that traverses through L . Alfred Tarski (1955) [61] showed that the fixed points of f in L form a complete lattice under \preceq . He further showed that if F is a set of order-preserving commutative functions, then the fixed points of all the functions in F form a complete lattice. He also presented its applications in set theory, Boolean algebra, topology and real functions.

Brouwer (1911) [62] showed that if f is a continuous function in a multidimensional closed simplex onto itself, then there exists a point x such that $f(x) = x$. Kakutani [63], further, provided the generalization of this theorem and studies a multidimensional closed simplex, where f is a point to set function f . He showed that if f is upper semi-continuous, then there exists a point x_0 such that $x \in f(x)$.

The fixed-point theorems have many applications. One of the applications is in the development of parallel processing algorithms that are tolerant to asynchrony, as we describe in Chapter 3, Chapter 4 and Chapter 5. Fixed-point theorem implies that all the optimal states, that belong to the same lattice L , form a lattice L' on their own. The infimum of L' is the optimal state that a system will converge to, if it is initialized in the infimum of L . A lattice-linear system that initializes in the infimum of a lattice is required to have at least one fixed point. A self-stabilizing system requires that the supremum is a fixed point. A silent self-stabilizing system requires that only the supremum is a fixed point.

In the next subsections, we present the applications of fixed point computation and how our results apply to them.

7.3.1 Conflict-free Replicated Datatypes

Introduced in [64], a *conflict-free replicated datatype* (CRDT) is a replicated data structure which can be accessed and modified by multiple processes. Each process has access to a distinct replica and the data structure is guaranteed to converge in a self-stabilizing manner even when these processes execute in asynchrony.

An example of such data structure is a vector for which the only allowed operation is to monotonically increase or decrease the values stored in it. The different states that this vector traverses through form a

\prec -lattice (\prec -lattice is called *semilattice* in [64]).

The theory of CRDTs has been extended to many other works. We note some of these works. The authors of [65] present an algorithm, as well as formal semantics, for a JSON data structure. This data structure automatically resolves concurrent modifications such that no updates are lost – all replicas converge towards the same state. A replicated set datatype is presented in [66]. The authors of [67] study integration of CRDTs with blockchain technologies to alleviate the additional latency between executing and committing transactions.

7.3.2 Fixed point iteration

In numerical analysis a *fixed point iteration* is the method of computing a fixed point. Essentially, given a function f with the same domain and codomain, and an element x_0 in the codomain of f , we input x_0 in f , and recursively provide the output of f as an input to it. Mathematically, we perform the recursion

$$x_{i+1} = f(x_i), \text{ for } i = 0, 1, 2, \dots$$

This iterative computation is performed until f stutters on a point, i.e., until a point x_l is found such that $f(x_l) = x_l$. In such a case, x is said to be a fixed point of f .

To find the least fixed point of a function f with respect to a lattice L , we start the above iterative computation from the infimum x_{inf} of L . Notice that this is how, e.g., the algorithm for stable marriage problem functions, to reach the optimal state.

The *least fixed-point* of a function f in a lattice L is the least element x in L for which $f(x) = x$. Similarly, the *greatest fixed-point* is the greatest element which is a fixed point for f . For example, in the lattice present in Figure 2.2, $\langle 1, 2, 2 \rangle$ is the least fixed-point of f_{smp} . To find the set of all fixed points of f with respect to a lattice, we initialize the above iterative computation starting from every point in a lattice.

Fixed point iteration is used by compilers for code optimization, e.g., through abstract interpretation [68] (where a compiler gains information about the semantics of a program – e.g., control flow, data flow – without performing all the computations). Apart from this, the fixed-point theorem has been used in several other fields such as economics, e.g., where John Nash introduced the Nash equilibrium [69] by exploiting the Kakutani fixed-point theorem (this theorem provides sufficient conditions for a function, whose domain and codomain are set-valued, that is defined on a convex, compact subset of Euclidean space to have a fixed point).

The vector in the PageRank algorithm [70] is a fixed point with respect to linear transformation, and by extension, all Markov chain models also search for a fixed point where they stabilize with respect to transition probability.

7.3.3 Modal μ -Calculus

μ -calculus is a logic that describes the properties of a transition system \mathcal{S} (cf. Section 2.1 for the definition of a transition system). A transition system can be an infinite graph. However, most transition systems, that we study in this dissertation, have finite size. We do not discuss the μ -calculus in detail, the reader is directed to [71] for an introduction on the topic. μ -calculus provides second-order expressive power which makes it extremely powerful logic in model checking. It utilizes the recursive computation of fixed-point iteration to express the operators of temporal logic.

7.3.4 Extending fixed-point logics to DAG-inducing systems

Observe that lattices and DAGs are special cases of simplices, where a global state, an n -dimensional tuple, can have one or more outgoing edges pointing to other global states. Thus, we have that a nondecreasing function whose domain is a set of n -dimensional tuples, forming a DAG, is a special case of the fixed point theorem by Kakutani [63]. Thus we have a corollary from Kakutani's theorem, which we state as follows.

Corollary 7.1. *(Of [63]) Let f be a nondecreasing point-to-set function whose domain and codomain is a finite set of n -dimensional tuples forming a DAG. Then, there is at least one point x such that $x \in f(x)$.*

7.4 Lattice-Linearity

In [6], the authors have studied lattice-linear problems which possess a predicate under which the states naturally form a lattice among all states. Problems like the stable marriage problem, job scheduling and others are studied in [6]. We study the theory established in [6] in detail in Section 2.3.

In [12], the authors have studied lattice-linearity in several dynamic programming problems.

The key idea of lattice-linearity is that a process/node determines that its local state is not feasible in any reachable optimal global state. In other words, it has to change its state to reach an optimal state. Thus, if node i changes its state from $st.i$ to $st'.i$ it never revisits state $st.i$ again. Consequently, the local states visited by a node form a total order. Hence, it can change its state even if it is relying on the old values of its neighbours. As a result, the nodes can run without synchronization and the system is guaranteed to reach an optimal state.

Garg, in [6], studied problems in which a distributive lattice is formed among the global state, where a meet and join can be found for any given pair of states, and meet and join distribute over each-other. However, we find that to allow asynchrony, a more relaxed data structure can be allowed. Specifically, in a \prec -lattice, for a pair of global states, their join can be found, however, their meet may not be found. For instance, in the instance that we study in Figure 5.2, both meet and join can be found for a pair of global states in a \prec -lattice, however, in the instance that we study in Figure 5.3, a join can be found for a pair of global states in a \prec -lattice but a meet is not always found.

Showing Property of Asynchrony for Existing Algorithms

Johnson’s algorithm for computing shortest paths [72] and Gale-Shapley algorithm for stable marriage [73] have been shown to be tolerant to asynchrony in [6]. The top trading cycle algorithm for housing market problem by Shapley and Scarf [74] (attributed to Gale) was shown to be lattice-linear in [11].

7.5 Problems Studied in this Dissertation

In this section, we discuss the related work on specific problems that we study in this dissertation. While doing so, we compare the properties of existing algorithms for these problems against the algorithms that we study in this dissertation. Note that in several cases, we do not develop new algorithms for problems, rather we show that an existing algorithm has better properties than originally proven, e.g., the guarantee of convergence in asynchrony.

Multiplication

In [8], the authors presented three parallel implementations of the Karatsuba algorithm for long integer multiplication on a distributed memory architecture. Two of the implementations have time complexity of $O(n)$ on $n^{\lg 3}$ processors. The third algorithm has complexity $O(n \lg n)$ on n processors.

We show that the Cesari-Maeder parallelization of the Karatsuba’s algorithm for multiplication is tolerant to asynchrony. This algorithm converges in $O(n)$ time. We also study a parallel processing algorithm for modulo operation, which is tolerant to asynchrony.

Modulo

In [75], the authors have presented parallel processing algorithms for inverse, discrete roots, or a large power modulo a number that has only small prime factors. A hardware circuit implementation for mod is presented in [76].

Dominating Set

Self-stabilizing algorithms for the minimal dominating set problem have been proposed in several works in the literature, for example, in [77–79]. Apart from these, the algorithm in [14] converges in $O(n^2)$ moves, and the algorithm in [15] converges in $9n$ moves under an unfair distributed scheduler. The best convergence time among these works is $4n$ moves.

We study a generalized version of minimal dominating set, the service demand based minimal dominating set problem, which is a more practical generalization of MDS than other algorithms present in the literature. We present an eventually lattice-linear self-stabilizing algorithm, converges in 1 round plus n moves (within $2n$ moves), and does not require a synchronous environment. In addition, evaluation of guards takes only $O(\Delta^4)$ time, which is better than the algorithm presented in [80]. We also study a fully lattice-linear self-stabilizing algorithm for minimal dominating set that converges in n moves and is fully tolerant to consistency

violations. These results present an improvement over the other algorithms present in the literature.

Vertex Cover

Self-stabilizing algorithms for the vertex cover problem has been studied in Astrand and Suomela (2010) [81] that converges in $O(\Delta)$ rounds, and Turau (2010) [82] that converges in $O(\min\{n, \Delta^2, \Delta \log_3 n\})$ rounds.

We present an eventually lattice-linear self-stabilizing algorithm for minimal vertex cover; it converges in 1 round plus n moves (within $2n$ moves). We also present a fully lattice-linear algorithm for minimal vertex cover that converges in n moves. These algorithms guarantee convergence in asynchrony.

Independent Set

Self-stabilizing algorithm for maximal independent set has been presented in [15], that converges in $\max\{3n - 5, 2n\}$ moves under an unfair distributed scheduler, [78] that converges in n rounds under a distributed or synchronous scheduler, [14] that converges in $2n$ moves.

We present an eventually lattice-linear self-stabilizing algorithm for maximal independent set; it converges in 1 round plus n moves (within $2n$ moves). We also present a fully lattice-linear algorithm for maximal independent set that converges in n moves. These algorithms guarantee convergence in asynchrony.

Graph Colouring

Self-stabilizing algorithms for graph colouring have been presented in several works, including [83–90]. The best convergence time among these algorithms is $n \times \Delta$ moves, where Δ is the maximum degree of the input graph.

We study an eventually lattice-linear self-stabilizing algorithm for graph colouring; it converges in $n + 4m$ moves. We also study a fully lattice-linear self-stabilizing algorithm for graph colouring; it converges in $n + 2m$ moves. These algorithms guarantee convergence in asynchrony.

2-Dominating Set

The 2-dominating set is not an extensively studied problem. The problem was introduced in [91]. A self-stabilizing algorithm for the 2-dominating set problem has been studied in [92]. This algorithm converges in $O(nD)$ rounds under a distributed scheduler, where D is the diameter of G .

We study an eventually lattice-linear self-stabilizing algorithm for 2-dominating set; it converges in 1 round plus $2n$ moves (within $3n$ moves), and is tolerant to asynchrony.

Robot Gathering on Discrete Grids

In a general case, it is impossible to gather a system of robots if their visibility graph is not a connected graph. One-axis agreement and distance-1 myopia are the minimal capabilities that robots need to converge on a triangular grid [16].

A system of robots with minimal capabilities has been studied with several output requirements, including gathering [16, 93, 94], dispersion [95], arbitrary pattern formation [96]. Gathering of robots has been studied

more recently in [3, 97]. We focus on systems of robots on grids, mainly the papers that study gathering.

Robots placed on an infinite rectangular grid were studied in [98], where the authors presented two algorithms for gathering. A synchronous scheduler is assumed and the robots require, respectively, distance-2 and distance-3 visibility. Moreover, under the latter algorithm, robots may not gather at one point but will gather a horizontal line segment of unit length.

Robots placed on an infinite triangular grid were studied in [99], where the authors provided an algorithm to form any arbitrary pattern. They require full visibility. Their algorithm works only when the the initial global state is asymmetric. Authors of [100] have studied gathering problem of 7 robots – initially, 6 of them form a hexagon and one robot is present at the centre of that hexagon. They require the initial state to form a connected visibility graph; the system finally reaches a global state where the maximum distance between two robots is minimized. A synchronous scheduler is assumed. In [93], authors characterized the problem of gathering on a tree and finite grid.

We study the algorithm presented by [16] and show that it can converge in asynchrony. This algorithm converges assuming that the robots are myopic, and can use a unidirectional camera, that sees one neighbour at a time. The robots form an arbitrary connected graph initially. Apart from the property of being able to converge in asynchrony, we also show that this algorithm converges in $2n$ rounds; the authors of the original algorithm showed that it converges in $2.5(n + 1)$ rounds.

Maximal Matching

A distributed self-stabilizing algorithm for the maximal matching problem is presented in [101]; this algorithm converges in $O(n^3)$ moves. The algorithm in [102] converges in $O(\log^4 n)$ moves under a synchronous scheduler. The algorithm for maximal matching presented in [103] converges in $n + 1$ rounds. Hedetniemi et al. (2001) [104] showed that the algorithm presented in [101] converges in $2m + n$ moves.

The PO-inducing algorithm for maximal matching, present in this dissertation, converges in $2n$ moves and is tolerant to asynchrony. This is an improvement to the algorithms present in the literature.

CHAPTER 8

CONCLUSION

Synchronization constraints affect how and when the nodes read data from each other. These constraints permit the user to design the algorithm at a high level and omit some of the details of the underlying system. Hence, they make the design of algorithms easier. However, enforcing synchronization creates an overhead that can lead to suboptimal use of computational resources. This is especially problematic, as we see a rise in the development and usage of large multiprocessor systems.

In this dissertation, we focused on developing algorithms that work correctly even without synchronization. We explore the necessary and sufficient properties of algorithms that allow them to execute without synchronization.

We showed that local state transitions being abstracted as a partial order is both necessary and sufficient for an algorithm to allow asynchrony. Focusing on the *necessary* condition, we find that if there is any existing algorithm that allows to be executed under asynchrony, then it induces a \prec -DAG in the global state space.

Organization of the Chapter

In Section 8.1, we discuss the specific contributions of this dissertation. In Section 8.2, we discuss how a PO-inducing algorithm can be transformed to other models. In Section 8.3, we discuss the practical applications and theoretical impact of this dissertation in multiprocessor systems technology. Finally, in Section 8.4, we discuss future work directions that arise from this dissertation.

8.1 Summary of Contributions

In this section, we summarise some key objective contributions of this dissertation.

Self-Stabilizing Lattice-Linear Problems

We observed the existence of lattice-linear problems that allow self-stabilization. We show that the parallel processing version developed by [8] for Karatsuba's multiplication algorithm (cf. [7]) has properties of asynchrony; we present one other algorithm for multiplication and two algorithms for the modulo operation that are lattice-linear, and thus, guarantee convergence in asynchrony. These algorithms are self-stabilizing, and in any given global state, all impedensable nodes can be identified.

Eventually Lattice-Linear Algorithms

We observed that eventually lattice-linear algorithms can be developed for non-lattice-linear problems. These algorithms induce one or more lattices only in a subset of the state space. They guarantee that starting from an arbitrary global state, the system traverses to a state in a lattice, and then traverses to an optimal state through that lattice. We develop eventually lattice-linear self-stabilizing algorithms for minimal dominating

set, minimal vertex cover, maximal independent set and graph colouring problems.

Fully Lattice-Linear Algorithms

We observed that fully lattice-linear algorithms can be developed for non-lattice-linear problems. Such algorithms induce lattices in the entire state space. We develop fully lattice-linear self-stabilizing algorithms for minimal dominating set, graph colouring, minimal vertex cover and maximal independent set problems. We also develop a lattice-linear 2-approximation algorithm for vertex cover; this algorithm is not self-stabilizing.

We showed for an algorithm developed in [16], that solves the gathering problem of myopic robots on an infinite triangular grid, that it is lattice-linear.

PO-Inducing Systems

We observed that the induction of a partial order in the local state transition graph is a necessary and sufficient condition to allow asynchrony. An observation that immediately follows is that since a total order is a special case of a partial order, all lattice-linear problems and algorithms are, respectively, PO-inducing problems and algorithms.

We showed that the dominant clique problem and the shortest path problem (where the path is required to be generated) are PO-inducing problems, and maximal matching is a non-PO-inducing problem. We present algorithms for these problems; the local state transition graph induced by these algorithms forms a partial order – it cannot be modelled within the constraints of a discrete structure such as the total order.

We derive time-complexity properties of a PO-inducing algorithm. As direct corollaries, we obtain the time-complexity properties of all the fully lattice-linear algorithms that we present in Chapter 5 and all the PO-inducing algorithms that we present in Chapter 6.

8.2 Transforming Our Algorithms to Other Models

In this dissertation, for the sake of simplicity, we presented algorithms that often require a node to read the values of its distance- x neighbours $x \geq 1$. For similar reasons, we present algorithms where nodes easily read data from each other as if they had direct access to the memory of each other. We note that these can be easily extended to other models of practical use while preserving the properties of interest. We identify some of these extensions, next.

Transforming to Distance-1

Some algorithms that we study in this dissertation require to read information about the nodes at distance- x from themselves, where $x > 1$. An algorithm, under which the nodes require to read information from other nodes at a high distance, can be costly to execute.

Since PO-inducing algorithms are tolerant to asynchrony and the nodes executing based on old information, a trivial transformation to distance-1 is to keep a copy of all variables of nodes in distance- x (cf. [17]). However, as we see in Section 5.6.1, a transformation that we present for Algorithm 5.2, a distance-4 al-

gorithm for minimal dominating set, we may not need to keep a copy of all such variables. As we see in Figure 5.4, the transformed algorithm has a substantial benefit in runtime as compared to the original algorithm.

Transforming to Message Passing Model

The algorithms that we study in this dissertation appear to execute such that the nodes can seamlessly read data from each other. As a result, these algorithms appear to be executing in the shared memory model. If the nodes, however, are placed remotely from each other, then the algorithm will need to be executed in the message passing model. We can very simply transform a shared memory PO-inducing algorithm into an algorithm that is executable in the message passing model. We discuss this in the following paragraph.

Since the algorithms we study are tolerant to asynchrony, we can assume, for message passing model, that every time a node changes its state, it announces its new state to other nodes. The target nodes may continue to perform executions based on old values until they receive this information. However, since the algorithm that they are executing is tolerant to asynchrony, their execution will not worsen the rank of the system.

8.3 Application and Impact of this Dissertation

In addition to applications in improving computing technology, this dissertation has an impact on the theory of multiprocessor systems and the way they are designed. Among such topics, we discuss some key areas in the following.

Designing a PO-inducing algorithm

We note that the techniques for PO-inducing algorithms are often different. However, a PO-inducing algorithm is closely related to the properties of the problem at hand. Additionally, the partial order imposed among states may involve auxiliary variables.

If while developing an algorithm, it is analyzed if it is PO-inducing, then a lot of assumptions about the properties of the system implementing that algorithm.

Being able to design systems that fully tolerate asynchrony has been a desired, albeit unattainable, objective. PO-inducing systems provide with a deterministic, discrete, guarantee to attain such fault tolerance.

Writing Proofs

One immediate implication of our theory is in its application in writing proofs. Developing systems that can tolerate asynchrony has been difficult, and one of the reasons that adds to the intricacy is writing proofs of correctness of such systems. Our theory not only insists on the possibility of simplifying such proofs, but also provides an upper bound to the time complexity of the runtime of asynchronous algorithms. Corollary 6.1 implies that to show that an algorithm is tolerant to asynchrony, we only need to show that the local states visited by individual nodes can be abstracted as a partial order, rather than to generate the entire state

space and checking for the existence of a cycle. Thus, our theory simplifies the detail that a proof would require in order to show tolerance to asynchrony of a multiprocessing system. In addition, Corollary 6.2 provides the upper bound of the time complexity of a PO-inducing algorithm.

Showing Property of Asynchrony for Existing Algorithms

Studying whether existing algorithms exploit lattice-linearity is extremely interesting and beneficial. Specifically, it allows us to eliminate costly, sophisticated, assumptions of synchronization from existing systems, instead of redesigning them from scratch.

A large number of problems can be solved by algorithms that may converge without requiring any synchronization. A high fraction of such algorithms may already be published, but it has not been proven so. Recently, there has been work that shows for some existing algorithms that they do not require synchronization. Such work includes the results present in this dissertation, where we show for existing algorithms that they do not require synchronization. We discuss this in the following paragraph.

We show tolerance to asynchrony in Cesari-Maeder parallelization [8] of Karatsuba’s multiplication and GSGS algorithm for converging myopic robots on an infinite triangular grid [16] in, respectively, Chapter 3 (Section 3.2.2) and Chapter 5 (Section 5.8).

There has been work by other authors who have proved for existing algorithms that they can be executed without synchronization. This work falls under the umbrella of PO-inducing algorithms. These results are listed in Related Work (Section 7.4).

The above results are a consequence of the observation that these algorithms stipulate that all impedensable nodes must update their local states, and that as a consequence, the local state transition graph forms a partial order.

8.4 Future Work

In the following, we discuss some interesting future work that can be extended from this dissertation.

Algorithm Design

One promising direction is to continue studying what other existing algorithms tolerate asynchrony. This will have an impact on alleviating the assumptions of synchronization from existing systems, and will thus, avoid having to design systems from scratch. Such work may have immediate applications in industrial computing technology.

Tolerating asynchrony does remove all requirements of synchronization, however, we have not seen any considerable improvement in the termination detection of such algorithms. Even these algorithms use the tools to detect termination, same as the algorithms that require synchronization. Despite numerous and ongoing efforts, developing a better termination detection protocol seems to be, as of yet, an unsolved problem. It would be very useful to bring about an improvement in the area of termination detection of

algorithms that execute without synchronization.

Systems Design

The existing algorithms that we have found results for, present a potential impact in discrete software engineering systems and operating systems. For example, we show for an existing parallel processing algorithm for multiplication (cf. Chapter 3) that it is tolerant to asynchrony. Most computational machines contain a circuit that computes the multiplication of a pair of bitstrings, which makes our result vital from the perspective of operating systems. Thus, along with theoretical problems and algorithms, a demanding and promising direction is to study algorithms that have applications in operating systems and firmware design. This will have an impact on improving the robustness of multicore, distributed and GPU systems. If a critical system is not straightforwardly tolerant to asynchrony, then it is worthwhile to investigate what minimal changes we can make to make it tolerant to asynchrony.

Another related research area is timeless circuits. Such circuits do not use a clock to organise the steps of its components, rather, the output of a component is fully dependent on the input fed to it by its predecessor component in the circuit (cf. Section 7.1). Such circuits have not received sufficient attention due to the scarcity of asynchronous algorithms, and the lack of any improvement in termination detection. Our research has a potential to motivate more substantial research in these areas; we expect our models and algorithms to perform even better when implemented on timeless circuits.

Machine Intelligence

Another application of our work is in machine learning, deep learning, and AI systems. We have recently started investigating such systems and found that many such systems can be divided into phases, where each individual phase can run asynchronously. However, when we merge those phases into one system, then it is required to put barriers between them. Specifically, transitioning between phases requires synchronization, and the current phase needs to be completed on all computing nodes before starting the next phase. Given the applications of such systems, a direction that is worth pursuing is to study if such systems can be allowed to run without synchronization between phases, and if not, then what minimal changes can be made to allow asynchrony.

BIBLIOGRAPHY

- [1] S. Cain, Bittersweet (Oprah’s book club). Crown Publishing Group, Apr. 2022.
- [2] O. W. Sacks, The man who mistook his wife for a hat and other clinical tales. New York, NY: Pocket Books, Apr. 1998.
- [3] S. Bhagat, S. Gan Chaudhuri, and K. Mukhopadhyaya, “Fault-tolerant gathering of asynchronous oblivious mobile robots under one-axis agreement,” in WALCOM: Algorithms and Computation (M. S. Rahman and E. Tomita, eds.), (Cham), pp. 149–160, Springer International Publishing, 2015.
- [4] L. Fang and P. Antsaklis, “Information consensus of asynchronous discrete-time multi-agent systems,” in Proceedings of the 2005, American Control Conference, 2005., pp. 1883–1888 vol. 3, 2005.
- [5] B. M. Assran, A. Aytakin, H. R. Feyzmahdavian, M. Johansson, and M. G. Rabbat, “Advances in asynchronous parallel and distributed optimization,” Proceedings of the IEEE, vol. 108, no. 11, pp. 2013–2031, 2020.
- [6] V. K. Garg, Predicate Detection to Solve Combinatorial Optimization Problems, p. 235–245. New York, NY, USA: Association for Computing Machinery, 2020.
- [7] A. Karatsuba and Y. Ofman, “Multiplication of many-digital numbers by automatic computers,” Doklady Ak- ademii Nauk SSSR, vol. 14, no. 145, pp. 293–294, 1962.
- [8] G. Cesari and R. Maeder, “Performance analysis of the parallel karatsuba multiplication algorithm for distributed memory architectures,” Journal of Symbolic Computation, vol. 21, no. 4, pp. 467–473, 1996.
- [9] J. E. Cohen, Food webs and niche space. Princeton University, 1978.
- [10] C. M. Chase and V. K. Garg, “Efficient detection of restricted classes of global predicates,” in Distributed Algorithms (J.-M. Hélary and M. Raynal, eds.), (Berlin, Heidelberg), pp. 303–317, Springer Berlin Heidelberg, 1995.
- [11] V. K. Garg, “A lattice linear predicate parallel algorithm for the housing market problem,” in Stabilization, Safety, and Security of Distributed Systems (C. Johnen, E. M. Schiller, and S. Schmid, eds.), (Cham), pp. 108–122, Springer International Publishing, 2021.
- [12] V. Garg, “A lattice linear predicate parallel algorithm for the dynamic programming problems,” in 23rd International Conference on Distributed Computing and Networking, ICDCN 2022, (New York, NY, USA), p. 72–76, Association for Computing Machinery, 2022.
- [13] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” Commun. ACM, vol. 21, p. 120–126, feb 1978.
- [14] S. Hedetniemi, S. Hedetniemi, D. Jacobs, and P. Srimani, “Self-stabilizing algorithms for minimal dominating sets and maximal independent sets,” Computers & Mathematics with Applications, vol. 46, no. 5, pp. 805–811, 2003.
- [15] V. Turau, “Linear self-stabilizing algorithms for the independent and dominating set problems using an unfair distributed scheduler,” Information Processing Letters, vol. 103, no. 3, pp. 88–93, 2007.
- [16] P. Goswami, A. Sharma, S. Ghosh, and B. Sau, “Time optimal gathering of myopic robots on an infinite triangular grid,” in Stabilization, Safety, and Security of Distributed Systems (S. Devismes, F. Petit, K. Altisen, G. A. Di Luna, and A. Fernandez Anta, eds.), (Cham), pp. 270–284, Springer International Publishing, 2022.
- [17] Y. Afek and S. Dolev, “Local stabilizer,” Journal of Parallel and Distributed Computing, vol. 62, no. 5, pp. 745–765, 2002.

- [18] D. Medhi and K. Ramasamy, Network Routing: Algorithms, Protocols, and Architectures. The Morgan Kaufmann Series in Networking, Morgan Kaufman, 2 ed., 2017.
- [19] C.-J. H. S. Janusz A. Brzozowski, Asynchronous Circuits. Monographs in Computer Science, Springer New York, NY, 1 ed., 1995.
- [20] M. Renaudin, “Asynchronous circuits and systems : a promising design alternative,” Microelectronic Engineering, vol. 54, no. 1, pp. 133–149, 2000.
- [21] C. Van Berkel, M. Josephs, and S. Nowick, “Applications of asynchronous circuits,” Proceedings of the IEEE, vol. 87, no. 2, pp. 223–233, 1999.
- [22] S. Furber, J. Garside, P. Riocreux, S. Temple, P. Day, J. Liu, and N. Paver, “Amulet2e: an asynchronous embedded controller,” Proceedings of the IEEE, vol. 87, no. 2, pp. 243–256, 1999.
- [23] S. Furber, “Computing without clocks: Micropipelining the arm processor,” in Asynchronous Digital Circuit Design (G. Birtwistle and A. Davis, eds.), (London), pp. 211–262, Springer London, 1995.
- [24] J. Garside, S. Temple, and R. Mehra, “The amulet2e cache system,” in Proceedings Second International Symposium on Advanced Research in Asynchronous Circuits and Systems, pp. 208–217, 1996.
- [25] H. van Gageldonk, K. van Berkel, A. Peeters, D. Baumann, D. Gloor, and G. Stegmann, “An asynchronous low-power 80c51 microcontroller,” in Proceedings Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems, pp. 96–107, 1998.
- [26] N. Paver, P. Day, C. Farnsworth, D. Jackson, W. Lien, and J. Liu, “A low-power, low noise, configurable self-timed dsp,” in Proceedings Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems, pp. 32–42, 1998.
- [27] S. Komori, H. Takata, T. Tamura, F. Asai, T. Ohno, O. Tomisawa, T. Yamasaki, K. Shima, H. Nishikawa, and H. Terada, “A 40-mflops 32-bit floating-point processor with elastic pipeline scheme,” IEEE Journal of Solid-State Circuits, vol. 24, no. 5, pp. 1341–1347, 1989.
- [28] H. Terada, M. Iwata, S. Miyata, and K. S., Advanced Topics in Dataflow Computing and Multithreading, ch. Superpipelined dynamic data-driven VLSI processors, pp. 75–85. IEEE Computer Society Press, 1995.
- [29] H. Terada, S. Miyata, and M. Iwata, “Ddmps: self-timed super-pipelined data-driven multimedia processors,” Proceedings of the IEEE, vol. 87, no. 2, pp. 282–295, 1999.
- [30] A. Abrial, J. Bouvier, M. Renaudin, P. Senn, and P. Vivet, “A new contactless smart card ic using an on-chip antenna and an asynchronous microcontroller,” IEEE Journal of Solid-State Circuits, vol. 36, no. 7, pp. 1101–1107, 2001.
- [31] J. Garside, S. Furber, and S.-H. Chung, “Amulet3 revealed,” in Proceedings. Fifth International Symposium on Advanced Research in Asynchronous Circuits and Systems, pp. 51–59, 1999.
- [32] A. Martin, A. Lines, R. Manohar, M. Nystrom, P. Penzes, R. Southworth, U. Cummings, and T. K. Lee, “The design of an asynchronous mips r3000 microprocessor,” in Proceedings Seventeenth Conference on Advanced Research in VLSI, pp. 164–181, 1997.
- [33] M. Renaudin and B. El Hassan, “The design of fast asynchronous adder structures and their implementation using dcvs logic,” in 1994 IEEE International Symposium on Circuits and Systems (ISCAS), vol. 4, pp. 291–294 vol.4, 1994.
- [34] M. M. Michael, “High performance dynamic lock-free hash tables and list-based sets,” in Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '02, (New York, NY, USA), p. 73–82, Association for Computing Machinery, 2002.

- [35] D. Hendler, N. Shavit, and L. Yerushalmi, "A scalable lock-free stack algorithm," in Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '04, (New York, NY, USA), p. 206–215, Association for Computing Machinery, 2004.
- [36] J. D. Valois, "Lock-free linked lists using compare-and-swap," in Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, PODC '95, (New York, NY, USA), p. 214–222, Association for Computing Machinery, 1995.
- [37] A. Natarajan and N. Mittal, "Fast concurrent lock-free binary search trees," in Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '14, (New York, NY, USA), p. 317–328, Association for Computing Machinery, 2014.
- [38] N. Shavit, E. Upfal, and A. Zemach, "A wait-free sorting algorithm," in Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing, PODC '97, (New York, NY, USA), p. 121–128, Association for Computing Machinery, 1997.
- [39] H. Attiya, N. Lynch, and N. Shavit, "Are wait-free algorithms fast?," J. ACM, vol. 41, p. 725–763, jul 1994.
- [40] D. Alistarh, K. Censor-Hillel, and N. Shavit, "Are lock-free concurrent algorithms practically wait-free?," J. ACM, vol. 63, sep 2016.
- [41] Y. Afek, D. Dauber, and D. Touitou, "Wait-free made fast," in Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing, STOC '95, (New York, NY, USA), p. 538–547, Association for Computing Machinery, 1995.
- [42] C. Dwork, M. Herlihy, and O. Waarts, "Contention in shared memory algorithms," J. ACM, vol. 44, p. 779–805, nov 1997.
- [43] J. Kim and A. Das, "Hcf: a starvation-free practical algorithm for maximizing throughput in input-queued switches," in HPSR. 2005 Workshop on High Performance Switching and Routing, 2005., pp. 68–72, 2005.
- [44] H. Attiya, V. Gramoli, and A. Milani, "A provably starvation-free distributed directory protocol," in Stabilization, Safety, and Security of Distributed Systems (S. Dolev, J. Cobb, M. Fischer, and M. Yung, eds.), (Berlin, Heidelberg), pp. 405–419, Springer Berlin Heidelberg, 2010.
- [45] W. H. Hesselink and M. IJbema, "Starvation-free mutual exclusion with semaphores," Formal Aspects of Computing, vol. 25, p. 947–969, Nov. 2013.
- [46] J. M. Morris, "A starvation-free solution to the mutual exclusion problem," Information Processing Letters, vol. 8, no. 2, pp. 76–80, 1979.
- [47] J. T. Udding, "Absence of individual starvation using weak semaphores," Information Processing Letters, vol. 23, no. 3, pp. 159–162, 1986.
- [48] J. Lejeune, L. Arantes, J. Sopena, and P. Sens, "A fair starvation-free prioritized mutual exclusion algorithm for distributed systems," Journal of Parallel and Distributed Computing, vol. 83, pp. 13–29, 2015.
- [49] S. Kanrar and N. Chaki, "Fapp: A new fairness algorithm for priority process mutual exclusion in distributed systems," Journal of Networks, vol. 5, Jan. 2010.
- [50] Y.-I. Chang, "Design of mutual exclusion algorithms for real-time distributed systems," J. Inf. Sci. Eng., vol. 11, no. 4, pp. 527–548, 1994.
- [51] D. Georgakopoulos, M. Rusinkiewicz, and A. Sheth, "On serializability of multidatabase transactions through forced local conflicts," in Proceedings. Seventh International Conference on Data Engineering, IEEE Comput. Soc. Press, 1991.

- [52] M. P. Fle and G. Roucairol, “On serializability of iterated transactions,” in Proceedings of the First ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, PODC ’82, (New York, NY, USA), p. 194–200, Association for Computing Machinery, 1982.
- [53] C. H. Papadimitriou, “The serializability of concurrent database updates,” J. ACM, vol. 26, p. 631–653, oct 1979.
- [54] N. Busi and G. Zavattaro, “On the serializability of transactions in javaspaces,” Electronic Notes in Theoretical Computer Science, vol. 54, pp. 92–105, 2001. ConCoord: International Workshop on Concurrency and Coordination (Workshop associated to the 13th Lipari School).
- [55] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues, “Making geo-replicated systems fast as possible, consistent when necessary,” in Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI’12, (USA), p. 265–278, USENIX Association, 2012.
- [56] P. Keane and M. Moir, “A simple local-spin group mutual exclusion algorithm,” IEEE Transactions on Parallel and Distributed Systems, vol. 12, no. 7, pp. 673–685, 2001.
- [57] A. Khanna, J. J. Rodrigues, N. Gupta, A. Swaroop, and D. Gupta, “Local mutual exclusion algorithm using fuzzy logic for flying ad hoc networks,” Computer Communications, vol. 156, pp. 101–111, 2020.
- [58] K. Raymond, “A tree-based algorithm for distributed mutual exclusion,” ACM Trans. Comput. Syst., vol. 7, p. 61–77, jan 1989.
- [59] J.-H. Yang and J. H. Anderson, “A fast, scalable mutual exclusion algorithm,” Distributed Computing, vol. 9, p. 51–60, Mar. 1995.
- [60] B. Knaster, “Un théorème sur les fonctions d’ensemble,” Annales de la Société Polonaise de Mathématique, vol. 6, pp. 133–134, 1928.
- [61] A. Tarski, “A lattice-theoretical fixpoint theorem and its applications,” Pacific Journal of Mathematics, vol. 5, no. 2, pp. 285 – 309, 1955.
- [62] L. E. J. Brouwer, “Über abbildung von mannigfaltigkeiten,” Mathematische Annalen, vol. 71, p. 97–115, Mar. 1911.
- [63] S. Kakutani, “A generalization of brouwer’s fixed point theorem,” Duke Mathematical Journal, vol. 8, Sept. 1941.
- [64] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, “Conflict-free replicated data types,” in Stabilization, Safety, and Security of Distributed Systems (X. Défago, F. Petit, and V. Villain, eds.), (Berlin, Heidelberg), pp. 386–400, Springer Berlin Heidelberg, 2011.
- [65] M. Kleppmann and A. R. Beresford, “A conflict-free replicated json datatype,” IEEE Transactions on Parallel and Distributed Systems, vol. 28, no. 10, pp. 2733–2746, 2017.
- [66] A. Deftu and J. Griebisch, “A scalable conflict-free replicated set data type,” in 2013 IEEE 33rd International Conference on Distributed Computing Systems, pp. 186–195, 2013.
- [67] P. Nasirifard, R. Mayer, and H.-A. Jacobsen, “Fabriccrdt: A conflict-free replicated datatypes approach to permissioned blockchains,” in Proceedings of the 20th International Middleware Conference, Middleware ’19, (New York, NY, USA), p. 110–122, Association for Computing Machinery, 2019.
- [68] P. Cousot and R. Cousot, “Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints,” in Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL ’77, (New York, NY, USA), p. 238–252, Association for Computing Machinery, 1977.
- [69] J. Nash, “Non-cooperative games,” The Annals of Mathematics, vol. 54, p. 286, Sept. 1951.

- [70] L. Page, S. Brin, R. Motwani, and T. Winograd, “The pagerank citation ranking: Bringing order to the web.,” Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.
- [71] J. Bradfield and C. Stirling, “12 modal mu-calculi,” in Handbook of Modal Logic (P. Blackburn, J. Van Benthem, and F. Wolter, eds.), vol. 3 of Studies in Logic and Practical Reasoning, pp. 721–756, Elsevier, 2007.
- [72] D. B. Johnson, “Efficient algorithms for shortest paths in sparse networks,” J. ACM, vol. 24, p. 1–13, jan 1977.
- [73] D. Gale and L. S. Shapley, “College admissions and the stability of marriage,” The American Mathematical Monthly, vol. 69, p. 9, Jan. 1962.
- [74] L. Shapley and H. Scarf, “On cores and indivisibility,” Journal of Mathematical Economics, vol. 1, no. 1, pp. 23–37, 1974.
- [75] T. Zeugmann, “Highly parallel computations modulo a number having only small prime factors,” Information and Computation, vol. 96, no. 1, pp. 95–114, 1992.
- [76] J. Butler and T. Sasao, “Fast hardware computation of $x \bmod z$,” in 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, pp. 294–297, 2011.
- [77] Z. Xu, S. T. Hedetniemi, W. Goddard, and P. K. Srimani, “A synchronous self-stabilizing minimal domination protocol in an arbitrary network graph,” in Distributed Computing - IWDC 2003 (S. R. Das and S. K. Das, eds.), (Berlin, Heidelberg), pp. 26–32, Springer Berlin Heidelberg, 2003.
- [78] W. Goddard, S. T. Hedetniemi, D. P. Jacobs, P. K. Srimani, and Z. Xu, “Self-stabilizing graph protocols,” Parallel Processing Letters, vol. 18, no. 01, pp. 189–199, 2008.
- [79] W. Y. Chiu, C. Chen, and S.-Y. Tsai, “A 4n-move self-stabilizing algorithm for the minimal dominating set problem using an unfair distributed daemon,” Information Processing Letters, vol. 114, no. 10, pp. 515–518, 2014.
- [80] H. Kobayashi, Y. Sudo, H. Kakugawa, and T. Masuzawa, “A Self-Stabilizing Distributed Algorithm for the Generalized Dominating Set Problem With Safe Convergence,” The Computer Journal, 03 2022. bxac021.
- [81] M. Åstrand and J. Suomela, “Fast distributed approximation algorithms for vertex cover and set cover in anonymous networks,” in Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA ’10, (New York, NY, USA), p. 294–302, Association for Computing Machinery, 2010.
- [82] V. Turau, “Self-stabilizing vertex cover in anonymous networks with optimal approximation ratio,” Parallel Processing Letters, vol. 20, no. 02, pp. 173–186, 2010.
- [83] A. Bhartia, D. Chakrabarty, K. Chintalapudi, L. Qiu, B. Radunovic, and R. Ramjee, “Iq-hopping: Distributed oblivious channel selection for wireless networks,” in Proceedings of the 17th ACM International Symposium on Mobile Ad Hoc Networking and Computing, MobiHoc ’16, (New York, NY, USA), p. 81–90, Association for Computing Machinery, 2016.
- [84] A. Checco and D. J. Leith, “Fast, responsive decentralized graph coloring,” IEEE/ACM Transactions on Networking, vol. 25, no. 6, pp. 3628–3640, 2017.
- [85] K. R. Duffy, C. Bordenave, and D. J. Leith, “Decentralized constraint satisfaction,” IEEE/ACM Transactions on Networking, vol. 21, no. 4, pp. 1298–1308, 2013.
- [86] K. R. Duffy, N. O’Connell, and A. Sapozhnikov, “Complexity analysis of a decentralised graph colouring algorithm,” Inf. Process. Lett., vol. 107, p. 60–63, jul 2008.

- [87] S. F. Galán, “Simple decentralized graph coloring,” Computational Optimization and Applications, vol. 66, pp. 163–185, Jan 2017.
- [88] D. J. Leith and P. Clifford, “Convergence of distributed learning algorithms for optimal wireless channel allocation,” in Proceedings of IEEE Conference on Decision and Control, pp. 2980–2985, 2006.
- [89] A. Motskin, T. Roughgarden, P. Skraba, and L. Guibas, “Lightweight coloring and desynchronization for networks,” in IEEE INFOCOM 2009, pp. 2383–2391, 2009.
- [90] D. Chakrabarty and P. de Supinski, On a Decentralized $(\Delta + 1)$ -Graph Coloring Algorithm, pp. 91–98. Symposium on Simplicity in Algorithms (SOSA), SIAM, 2020.
- [91] B. Bollobas, E. J. Cockayne, and C. M. Mynhardt, “On generalised minimal domination parameters for paths,” Discrete Mathematics, vol. 86, pp. 89–97, 1990.
- [92] S. Maruyama, Y. Sudo, S. Kamei, and H. Kakugawa, “A self-stabilizing 2-minimal dominating set algorithm based on loop composition in networks of girth at least 7,” in 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, May 2022.
- [93] G. D’Angelo, G. Di Stefano, R. Klasing, and A. Navarra, “Gathering of robots on anonymous grids and trees without multiplicity detection,” Theoretical Computer Science, vol. 610, pp. 158–168, 2016. Structural Information and Communication Complexity.
- [94] P. Flocchini, G. Prencipe, N. Santoro, and P. Widmayer, “Gathering of asynchronous robots with limited visibility,” Theoretical Computer Science, vol. 337, no. 1, pp. 147–168, 2005.
- [95] J. Augustine and W. K. Moses, “Dispersion of mobile robots: A study of memory-time trade-offs,” in Proceedings of the 19th International Conference on Distributed Computing and Networking, ICDCN ’18, (New York, NY, USA), Association for Computing Machinery, 2018.
- [96] K. Bose, R. Adhikary, M. K. Kundu, and B. Sau, “Arbitrary pattern formation on infinite grid by asynchronous oblivious robots,” Theoretical Computer Science, vol. 815, pp. 213–227, 2020.
- [97] R. Klasing, A. Kosowski, and A. Navarra, “Taking advantage of symmetries: Gathering of many asynchronous oblivious robots on a ring,” Theoretical Computer Science, vol. 411, no. 34, pp. 3235–3246, 2010.
- [98] P. Poudel and G. Sharma, “Time-optimal gathering under limited visibility with one-axis agreement,” Information, vol. 12, no. 11, 2021.
- [99] S. Cicerone, A. Di Fonso, G. Di Stefano, and A. Navarra, “Arbitrary pattern formation on infinite regular tessellation graphs,” in Proceedings of the 22nd International Conference on Distributed Computing and Networking, ICDCN ’21, (New York, NY, USA), p. 56–65, Association for Computing Machinery, 2021.
- [100] M. Shibata, M. Ohyabu, Y. Sudo, J. Nakamura, Y. Kim, and Y. Katayama, “Gathering of seven autonomous mobile robots on triangular grids,” in 2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 566–575, 2021.
- [101] S.-C. Hsu and S.-T. Huang, “A self-stabilizing algorithm for maximal matching,” Information Processing Letters, vol. 43, no. 2, pp. 77–81, 1992.
- [102] M. Hanckowiak, M. Karonski, and A. Panconesi, “On the distributed complexity of computing maximal matchings,” SIAM Journal on Discrete Mathematics, vol. 15, no. 1, pp. 41–57, 2001.
- [103] W. Goddard, S. Hedetniemi, D. Jacobs, and P. Srimani, “Self-stabilizing protocols for maximal matching and maximal independent sets for ad hoc networks,” in Proceedings International Parallel and Distributed Processing Symposium, pp. 14 pp.–, 2003.

- [104] S. T. Hedetniemi, D. P. Jacobs, and P. K. Srimani, “Maximal matching stabilizes in time $o(m)$,” Information Processing Letters, vol. 80, no. 5, pp. 221–223, 2001.
- [105] A. T. Gupta, “Burning geometric graphs,” Master’s thesis, Indian Institute of Information Technology Vadodara, 2020.
- [106] L. Sigler, Fibonacci’s Liber Abaci. Springer New York, 2002.
- [107] A. Blair, P. Duguid, A.-S. Goeing, and A. Grafton, eds., Information. Princeton, NJ: Princeton University Press, Jan. 2021.
- [108] S. Dasgupta, It Began with Babbage: The Genesis of Computer Science. Oxford University Press, feb 2014.
- [109] L. F. Menabrea, Scientific Memoirs, vol. 3, ch. Sketch of the analytical engine invented by Charles Babbage, Esq, pp. 666–731. Taylor and Francis, 1843.
- [110] A. M. Turing, “On computable numbers, with an application to the entscheidungsproblem,” Proceedings of the London Mathematical Society, vol. s2-42, no. 1, pp. 230–265, 1937.
- [111] M. R. Garey and D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness. USA: W. H. Freeman & Co., 1979.
- [112] Wikipedia, “The free encyclopedia.”

APPENDIX A

OVERVIEW OF GRAPH THEORY AND DISTRIBUTED ALGORITHMS

The intention behind adding this appendix is to assist a reader, who is new to distributed systems and graph theory, in understanding the concepts on which this dissertation is based. However, we assume that a reader is acquainted with Boolean algebra. Part of Appendix A.1 and Appendix A.2 is taken verbatim from Burning Geometric Graphs (A T Gupta, Master’s Thesis, 2020) [105].

A.1 Preface to Algorithms

Lionardo Pisano, more popularly known as Fibonacci, was an Italian mathematician, and he introduced the conventional Indian mathematical methods to Europe in the 13th century [106]. In his book, *Liber Abaci* (liberation from abacuses/abaci), he introduces *modus Indorum* (method of the Indians). Until then, the abacus was used across Europe to perform mathematical calculations. Pisano introduced a mathematics which was more efficient: computations could be performed on numbers without bounds on their digit length.

Earlier than Pisano, Muhammad ibn Musa al-Khwarizmi, a Persian mathematician, in 9th century, wrote *kitāb al-hisāb al-hindī* (book of Indian arithmetic) and *kitāb al-jam’ wa’l-tafriq al-hisāb al-hindī* (book of addition and subtraction in Indian arithmetic). A few centuries later, Al-Khwarizmi’s texts were translated to Latin [107].

It is due to the work of Al-Khwarizmi and Pisano that the methods of Indian arithmetic spread across Europe, and a person who could perform computations without the use of abacus was called *Maestro-de-abaci*. After a series of nomenclatural adaptations, the Europeans started to call this new form of mathematics, which could be performed without abacus and without bounds on the input size, *algorithms*.

Since then, numerous efforts have been made to translate human intelligence and computing ability into artificial machinery. Blaise Pascal [108] built a machine in the 17th century which could perform addition and subtraction. Gottfried Wilhelm Leibniz built a machine, during the same time, which could perform multiplication and division as well. Charles Babbage built the famous *Difference Engine* which could do similar computations *automatically*, that is, once the input numbers are supplied to it, it was able to do the computation without any human intervention. This machine was able to prepare tables: it was able to compute polynomials of degree 2 for consecutive integers; this was called the *method of differences*. Babbage built the first prototype of this machine in 1822.

Luigi Frederico Menabrea explained with reference to the Difference Engine that it was limited only to one type of computations, it could not be applied to solve numerous other problems in which mathematicians might be interested. This led Babbage to design the *Analytical Engine*, which could solve the full range of algebraic problems. The generality of the Analytical Engine is discussed in Menabrea’s Italian article *Sketch*

of the *Analytical Engine* (1842), which was translated into English by Augustus Ada [109].

Augustus Ada, countess of Lovelace, proposed that Babbage's design could be used to compute function of any number of functions. On Babbage's request, she wrote some additional notes to her *memoir*, most famous one of them is the *Note G*, in which, firstly, she anticipated an issue: whether computers can exhibit *intelligence*, or, *original thought*, and secondly, in this note she wrote a sequence of operations (an *algorithm*) to compute Bernoulli numbers on the Analytical Engine.

Modern definition of the word algorithm is as follows. An *algorithm* is a step-by-step procedure used to solve a problem given that it halts in finite time for any given input.

After some decades, Alan Mathison Turing worked on decision functions (or a decision problem, as he presents in [110]) and their representations. A decision function is a sequence of mathematical instructions whose outputs are *accept* or *reject* on an arbitrary input string. A decision function solves a decision problem. The set of strings which a function accepts is the *language* of that function. This set of strings defines the problem which that function solves.

Turing initiated the design of what we call the Turing Machine which works on these formal languages to compute for any decision problem. We do not discuss the Turing Machine; the reader is advised to refer [110,111] to study the Turing Machine in detail. We start with a brief discussion on graphs, on which the chapters in this dissertation are majorly based.

A.2 Preface to Graphs

A *graph* is a representation of entities and their relations: generally, a graph tells which entities are related (unweighted graph); sometimes the relations may have some associated cost or weightage (weighted graphs). Formally, a graph is a mathematical object which represents entities as vertices, and relations as edges between those vertices: if two entities are related, then there will be an edge between their corresponding vertices in the graph. The number of vertices in a graph is its *order*, and the number of edges in a graph is its *size*. Generally, given a graph, all its edges represent the same type of relation.

An example of an unweighted graph is presented in Figure A.1. In this graph, 1, 2, 3 and 4 are the vertices and, for example, there is an edge between vertices 1 and 2. This graph can be represented by the sequence $G_4^1 = \langle 4, 7, 8, 11, 13, 2, 1, 3, 4, 2, 4, 2, 3 \rangle$. The first element in G_4^1 , 4, represents the total number of vertices. The second element of G_4^1 , 7, represents that the vertices connected to the first vertex start at position 7 in G_4^1 . Similarly, the third element of G_4^1 , 8, represents that the vertices connected to the second vertex start at position 8 in G_4^1 . So the first vertex, vertex 1, is only connected to vertex 2, and vertex 2 is connected to vertices 1, 3 and 4. Similarly, the fourth and fifth elements of G_4^1 , 11 and 13, represent that the vertices connected to the third and fourth vertices start at position 11 and 13 in G_4^1 respectively. The sixth element, 15, represents that G_4^1 contains only 14 elements.

A graph in which all the edges are of equal weight is also viewed as an unweighted graph. Generally, in the graphs that are deemed as unweighted graphs, all edges are represented with the weight value 1. In the graph represented in Figure A.1 also, we consider the weight on all edges to be 1, which we do not show explicitly for brevity.

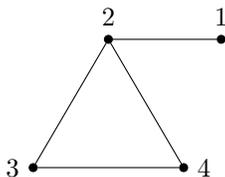


Figure A.1: A sample graph represented by the sequence $G_4^1 = \langle 4, 7, 8, 11, 13, 15, 2, 1, 3, 4, 2, 4, 2, 3 \rangle$.

Some of the graphs that we study in this dissertation are weighted graphs. A *weighted graph* is a graph that has a weight associated to the edges that connect the vertices. A weighted edge represents a cost or weightage of a relation between a pair of entities. Such cost-associated, or weighted, edges signify some real-world properties. For example, if two computers are connected in a network, we can model them as a pair of vertices, and there will be an edge connecting those vertices. The frequency of communication between them can be modelled by assigning that much weight to that edge. A graph constructed in this way will provide information about the frequency of communication among computers in a network. For another example, a weighted edge joining a pair of vertices (where these vertices represent two cities) can represent that there is a direct road connecting those cities, and its weight would represent the distance between their corresponding cities. In the above computer network example, the edges represent weightage (priority) of the connections, whereas in the city example, the edges would represent the cost of the connections.

We also study directed graphs. In an undirected graph, we have edges *directed* both to and from the vertices that they connect. So, an undirected edge joining two vertices a and b has directions both from a to b and from b to a . Thus, an undirected edge is effectively a bidirectional edge, as it can be depicted from the description of G_4^1 and its representation in Figure A.1. A *directed graph* is a graph in which the edges are unidirectional. A *mixed graph* is a graph that contains both unidirectional and bidirectional edges.

An example of a mixed weighted graph is presented in Figure A.2. In this graph, 1, 2, 3 and 4 are the vertices and, for example, there is an edge from vertex 1 to vertex 2. This graph can be represented by the sequence $G_4^2 = \langle 4, 7, 8, 8, 10, 12, \langle 2, 1 \rangle, \langle 2, 3 \rangle, \langle 4, 5 \rangle, \langle 2, 4 \rangle, \langle 3, 5 \rangle \rangle$. The first element in G_4^2 , 4, represents the total number of vertices. The second element of G_4^2 , 7, represents that the vertices that have edges from vertex 1 start at position 7 in G_4^2 . Similarly, the third element of G_4^2 , 8, represents that the vertices that have edges from vertex 2 start at position 8 in G_4^2 . So vertex 1 has an edge to vertex 2 only; the weight of this edge is 1. Similarly, the fourth and fifth elements of G_4^2 , 8 and 10, represent that the vertices that

have edges from vertex 3 and vertex 4 start at position 8 and 10 in G_4^2 respectively. The sixth element, 12, represents that G_4^2 contains only 11 elements. Since the start index is 8 corresponding to both vertex 2 and vertex 3, we have that there is no edge that goes from vertex 2 to any other vertex. The edge between vertex 3 and vertex 4 is a bidirectional edge, which can be represented as having no arrows, as in Figure A.2, or having arrows on both its ends.

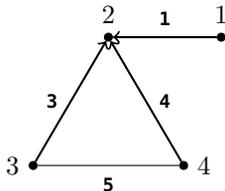


Figure A.2: A sample graph represented by the sequence $G_4^2 = \langle 4, 7, 8, 8, 10, 12, \langle 2, 1 \rangle, \langle 2, 3 \rangle, \langle 4, 5 \rangle, \langle 2, 4 \rangle, \langle 3, 5 \rangle \rangle$.

Note that we do not need to represent the edges of a weighted graph as a two-value tuple. We can represent such a graph as a single dimension tuple, where each edge will occupy two consecutive indices in that tuple, where the first of those indices represents the direction of the edge and the second index represents its weight. For example, G_4^2 can also be represented as $G_4^3 = \langle 4, 7, 9, 9, 13, 17, 2, 1, 2, 3, 4, 5, 2, 4, 3, 5 \rangle$

In a directed graph, a *supremum* of a pair vertices a and b is the closest vertex from a and b that has a path from both a and b . Similarly, an *infimum* of a and b is the closest vertex to a and b that has a path to both a and b .

A *complete lattice* is a directed graph in which, for every pair of vertices a and b , there exists a unique infimum and supremum of a and b . As an example, consider a directed graph in which the vertices are natural numbers, and there is an edge from a to b iff b is equal to a , multiplied by a prime number. Notice that this directed graph is a complete lattice: for every pair of numbers a and b , there is a unique supremum (which is the lowest common multiple (LCM) of a and b) and there is a unique infimum (which is the highest common factor (HCF) of a and b). We call this lattice a *prime-factorizability lattice*. We have shown a subgraph of the prime-factorizability lattice for numbers from 1 to 20 in Figure A.3. In a prime-factorizability lattice, a number a is a factor of b iff there is a path from a to b . Note that the prime-factorizability lattice is of infinite order and size. The supremum of, for example, 16 and 20 is 80, which is not shown in Figure A.3.

In this dissertation, the lattices that we study have a supremum defined for any given pair of vertices, however, an infimum may not be found. Such lattices come within the class of *incomplete lattices*.

The prime-factorizability lattice can be used to study several properties of numbers, for example: (1) iff there is a path from a to b , then a is a factor of b , (2) iff b is a multiple of a , then the supremum of a and b is b , and the infimum of a and b is a , (3) iff the infimum of a pair of numbers a and b is 1, then a and b are

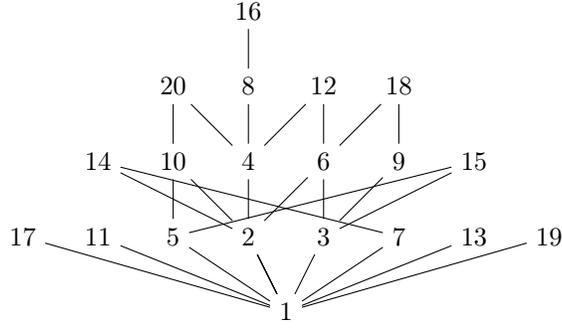


Figure A.3: A subgraph of the prime-factorizability lattice for natural numbers from 1 to 20. All edges are directed upwards; arrows are not shown for brevity.

coprime, and (4) if p is a prime number, then p is connected to 1 by a direct edge.

Another example lattice is shown in Figure A.4. In this lattice, each vertex is a tuple of three numbers. There is an edge from tuple a to tuple b iff (1) all elements of b (sequentially) are equal to or greater than the elements of a , that is, $\forall i, 1 \leq i \leq 3 : b[i] \geq a[i]$, (2) all but one elements (sequentially) of a and b are different, that is, $\exists i, 1 \leq i \leq 3 : (a[i] \neq b[i] \wedge (\forall j \neq i : a[j] = b[j]))$, and (3) at the index i where a and b are different, $b[i] - a[i] = 1$. Notice that the graph shown in Figure A.4 is a complete lattice.

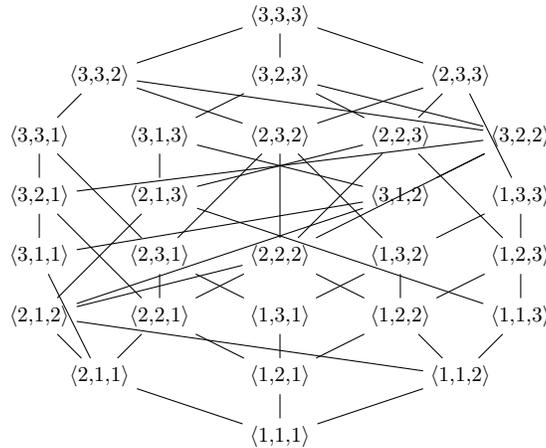


Figure A.4: An example lattice where each vertex is a tuple. All edges are pointing upwards. We have not shown the arrows for brevity.

A.3 Transitive Edges

Let that R be a relation and for a pair of objects a and b , let aRb be true if and only if b is related to a under R . We say that R is a *transitive relation* only if the following is true: for any three objects, a , b and c , if aRb is true and bRc is true, then aRc is also true.

As an illustration of a relation that a graph can represent, consider the lattice presented in Figure A.4. Let that each tuple in this lattice represents some state of a system. Let this lattice be \mathcal{L} . We know that

all the edges in this lattice are pointing upwards. An algorithm A that follows this lattice can take a system from a state s_1 to another state s_2 in one step only if there is an edge from s_1 to s_2 . If we consider a relation represented by a graph such as \mathcal{L} , then we have that s_2 is related to s_1 if and only if there is an edge from s_1 to s_2 . It also means that s_2 is related to s_1 if and only if A is able to take the system from s_1 to s_2 in one step.

In this dissertation, at several places, we discuss transitive edges. If we impose transitivity in \mathcal{L} , then it would mean we are considering \mathcal{L}' instead of lattice \mathcal{L} , such that there are edges from a state s_1 to another state s_2 in \mathcal{L}' if and only if there is a path from s_1 to s_2 in \mathcal{L} . An algorithm A' , that follows the lattice \mathcal{L}' , will be able to transition the system from s_1 to s_2 in \mathcal{L}' in one step only if there is a path from s_1 to s_2 in \mathcal{L} . We discuss more on transitivity, and some applications transitivity in directed graphs, in Chapter 2 and the chapters that follow.

A.4 Preface to Parallel Processing

The soul of this dissertation lies in methods of parallel processing. Contemporary examples of parallel processing systems are a graphical processing unit (GPU) or a cluster of computers connected to each other via Ethernet, or computers even more remotely connected to each other, where, between a pair of computers, several channels can be involved, e.g., one or more of an Ethernet cable, WiFi, optical cable.

When a single processor solves a problem, then it may solve that problem in say, k time units. But when we use a system comprising several processors, then we can distribute the work among those processors. We already have a single processor powerful enough, now one would expect that a bunch of, say, n processors would be n times as powerful. However, there are other costs that a parallel processing system must pay in order to utilize such increased power due to parallelization. Let us consider the following example, an instance of job distribution in a kitchen rather than a multiprocessor system.

Let us assume that a cook takes 2 hours to prepare a certain dish. Let us assume that this dish does not require any cooking, and only mixing certain vegetables and spices, but the vegetables have to be prepared and sliced, each, in a certain way which takes this much time in total. Now let us assume that we have 3 cooks at our disposal. Here, the cooks are the job-doers. In the same way, in a computer system, a processor is the job-doer. 3 cooks might take about 40-45 minutes to prepare this dish, as they can prepare all the ingredients consequently and then mix them at the end. Taking 40-45 minutes is reasonable. Now assume that there are 100 cooks. Ideally, they should take less than 2 minutes altogether to prepare the dish. However, all those cooks should communicate their states in terms of progress and the part of the preparation they want to take up. Now on average, if I am one of those cooks, I will take about a minute and a half to talk to one other cook, and about 150 minutes to talk to all other cooks. Similarly, all cooks would want to talk to all other cooks, maybe more than once. This is going to take time even more than

what a single cook would take to prepare the dish. So most of the power of the job-doers in our kitchen is consumed in communication for synchronization.

As we assumed above, a single processor would take k time units to compute for a problem. But if we use n processors, it may take more than $\frac{k}{n}$ time units to solve the same problem, even when the work is evenly divided. The extra time that they take is invested in communication for synchronization. Similar to cooks in our kitchen example, processors need to communicate with each other and they need to be synchronized with each other.

This dissertation investigates the behaviour of the systems where we eliminate the need for synchronization among the processors, and thereby, the costs incurred to enforce it. Some problems naturally allow asynchrony, while in other problems, we have to add details to the executions algorithmically.

For example, we could develop an algorithm for these 100 cooks and associate their names with the part of preparation that they have to take care of. Then they would simply work asynchronously and get their parts done, and then mix the ingredients at the end. Depending on the vegetables, it would take roughly two minutes to cut them precisely and mix them all with the required spices. Now any cook could choose any part of the preparation, other than what he is assigned, so this *problem* does not allow asynchrony naturally, and therefore we must *impose* on them their respective jobs *algorithmically*. We do come across such problems in computer science as well.

There are, however, problems that naturally allow asynchrony. For example, consider that an aquarium company of pre-telephonic times, established near a freshwater lake, requires 100 pounds of live fish to put in its aquariums. Assume that a ship can collect 10 pounds of fish in a day. So it would take 10 days for a single ship to collect that amount of fish. If that aquarium company owns 5 ships, each ship can be sent off with the target of 20 pounds of fish each. This job can be done in roughly 2 days. There is no job association with, e.g., the name of the ship; all ships have to catch a specific, same, amount of fish, so such a system *naturally* allows asynchrony.

Now assume that we have not assigned the amount of fish to all boats and require them to return, all, at the same time. One might think that this requirement will take the least possible time for all ships to come back, but it is not so. In the above system, the ships might have returned to port at different times, because they may be working at different speeds in catching the fish. However, in this latter case, we will have to utilize synchronization, so that we can ensure that the ships stay in communication with each other regarding the amount of fish each of them has to catch, and return to port at the same time. But since these are pre-telephonic times, exchanging information through longboats, will take considerable time in itself and much time will be consumed only in waiting for information to arrive.

In this dissertation, we study the characteristics of problems that allow asynchrony naturally, and the

structure of executions of algorithms that impose asynchrony in solving problems that do or do not allow asynchrony naturally.

A.5 Preface to Acyclic State Transitions

Above, we discussed the amount of resources and time used to enforce synchronization in a kitchen and a fish-catching example. In a multiprocessor system, where multiple processors perform execution to solve a problem, resources and time are invested, with a similar proportion, to enforce synchronization. In this dissertation, by the word *system* we refer to a multiprocessor system.

The progress made by a system can be evaluated by analyzing its *global state*, which is represented by the values stored in variables throughout all the processes. For now, we will call the state of convergence to be a global state where the system is deemed to have solved the problem at hand, and the solution is represented by the state that the system is in (by the values stored in variables throughout all the processes when the system is in the state of convergence). If a system is not synchronized, then it can potentially make mistakes, which may take it *farther away* from the state of convergence. Synchronization prevents these mistakes from happening. Furthermore, the assumption of synchronization restricts when the nodes will read from other nodes and take action, and consequently, it makes the design of algorithms easy.

There are different types of synchronization, which we describe at various places throughout this dissertation. The type of synchronization that a system needs to enforce depends on the nature of the problem and the structure of the system. An absence of an apt synchronization primitive, where it would be otherwise required, can make the system make mistakes. These mistakes are caused because of executions that the processes make based on old and inconsistent information. We call such faults *consistency violation faults*. Such mistakes result in the system to commit cyclic state transitions, where a system makes some progress, but then moves further away from the state of convergence because of a consistency violation. This can repeat continually, and everytime the system makes progress, it will again move farther away from the state of convergence; this happens due to the absence of an apt synchronization primitive, even if the algorithm is otherwise correct on a uniprocessor system.

Proper synchronization ensures that an acyclic structure is induced among the state transitions, as it ensures that the data flows among the processes in a consistent fashion.

In this dissertation, we study the properties of systems that guarantee convergence without synchronization. The essential property for an algorithm to allow asynchrony is that it should be able to enforce acyclic state transitions in the system. There are multiple processors in a system running at least one process each. Acyclic state transitions can be enforced even in asynchrony as follows. Let us assume that every process performs execution such that a local state once discarded is never revisited again (a *local state* of a process is represented by the values of only its own variables). However, the processes need to make such transitions

carefully: they must ensure that if they are discarding their local state, then that local state is infeasible for any possible desired state of convergence. Enforcing such guarantees is problem-dependent; this, we explore throughout this dissertation. If such guarantees can be made, then even if a node is reading old values, it can safely make a transition if it decides so, because its current local state would be infeasible for any desired state of convergence. Consequently, the system exhibits acyclic state transitions, which is the main subject of exploration for this dissertation.

A.6 An Example Graph Theoretic Problem

Consider, e.g., a fort containing three *indexed* towers; each pair of towers has a straight path connecting them. This is demonstrated in Figure A.5. Consider that the fort is under attack. All three paths are well protected with high and strong walls. The only entry point to enter the fort are the towers. The problem, thus, is to place archers *minimally*; this problem is not of how many archers to place, the problem is *where* to place them. We can place archers on all three towers, any two towers or just one tower. Notice that if we place archers in only one of the towers, they can protect the tower that they are placed in, as well as the other two towers. Therefore, for this problem, we have that placing one or more archers in only one tower is sufficient to protect all three towers, and hence the entire fort.

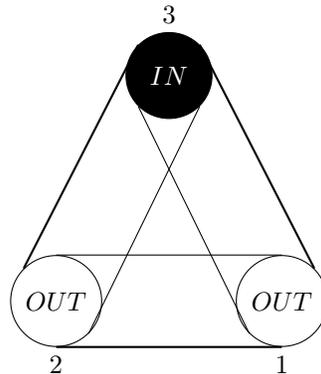


Figure A.5: Fort under attack: archers positioned at the tower marked *IN*.

The threads in a multiprocessing system have a distinct ID associated to them. An algorithm for this problem computes a *minimal* set of towers such that all towers can be protected. An example algorithm for such a problem makes each processor simulate a distinct tower. We call a processor, that performs computation in a multiprocessor system, a computation *node*. A node can be in states *IN* or *OUT*; if a node is *IN*, then it means that the algorithm decided to place archers in its corresponding tower, or otherwise, if a node is *OUT*, then it means that the algorithm decided *not* to place any archers in its corresponding tower.

Consider an algorithm in which the nodes check if one of their neighbours is *IN*; if none of their neighbours

are *IN*, then they will *move IN*. Suppose that initially, all nodes are *OUT*. Now let that node 2 and node 3 execute together. They will *see* that their neighbouring nodes are *OUT*, so both node 2 and node 3 will change their state to *IN*. In such an output, notice that the fort is protected, but not minimally. Thus, to allow only a minimal set of nodes to move in, we need some synchronization primitive to be deployed among the nodes. Most algorithms in the literature, that are developed for multiprocessor systems, assume some synchronization primitive to be deployed among the nodes.

In the paragraph above, we discussed an algorithm for minimal dominating set problem where the computing nodes should not be allowed to run in asynchrony. Now consider another algorithm for this problem, in which, if a set of nodes want to move in, they will allow the node with a higher ID in their neighbourhood to move first, and wait until then. In such an algorithm, node 3 will move (change its state to) *IN*. Node 1 and node 2 will wait for node 3 to move first. After when node 3 moves *IN*, when node 1 and node 2 evaluate their required *action*, they will see that they are already being protected by node 3 (tower 3), so they will not perform any move (i.e., they will not change their state). Thus, we have a minimal set of towers that are able to protect the entire fort. Notice that this algorithm, unlike the algorithm described in the above paragraph, can be allowed to run asynchronously for this problem. This is because we used a tie-breaker based on node IDs, so *race conditions* like the previous algorithm do not evolve. Thus, Figure A.5 shows the output of this algorithm, where we obtain a minimal set of locations to place the archers. In this dissertation, we study the properties of such algorithms: algorithms that can be allowed to run without synchronization.

The problem described above is the dominating set problem. In the *dominating set* problem, the input is a graph G , and the task is to compute a dominating set \mathcal{D} such that every node i should be *dominated*: (1) either i is *in* the dominating set, or (2) at least one of the neighbours of i is *in* the dominating set. We consider all and only the nodes that are *IN* to be in \mathcal{D} . In the *minimal dominating set* problem, the input is a graph G , and the task is to compute the dominating set problem on G and compute \mathcal{D} such that if any node is removed from \mathcal{D} , then some node in G becomes not dominated. The dominating set problem solves the problem of *domination*, whereas the minimal dominating set problem solves the problem of *minimality* along with domination.

Assume that in the initial state, all the nodes that represent the three towers are *OUT*. With such an input setting, notice that in the two algorithms that we described in the above paragraphs, the former algorithm solves the dominating set problem, even if it is run in asynchrony. However, it can solve the minimal dominating set problem only if it uses synchronization (in this case, local mutual exclusion or a central scheduler). The latter algorithm solves the minimal dominating set problem with or without synchronization.

In this dissertation, we study the properties that enable an algorithm to guarantee convergence in asyn-

chrony. We study several problems for which such algorithms can be developed, and present example algorithms. We also study example problems where some specific algorithm designs do not work, and what algorithm design approaches would work. In such cases, we also study how the properties of the subject problems are responsible for certain algorithm design approaches to not work, and how these properties are correlated to the design approaches that work.

APPENDIX B

PUBLICATIONS FROM THIS DISSERTATION

We wrote the following peer-reviewed papers, published until the date of the defence, that emerged from this dissertation.

- [6] Gupta, A. T. and Kulkarni, S. S. (2024) Tolerance to Asynchrony of an Algorithm for Gathering Myopic Robots on an Infinite Triangular Grid. In Proceedings of the 19th European Dependable Computing Conference. EDCC 2024.
- [5] Gupta, A. T. and Kulkarni S. S. (2024) Eventually Lattice-Linear Algorithms. Journal of Parallel and Distributed Computing. (extended version of conference paper [1])
- [4] Gupta, A. T. and Kulkarni, S. S. (2023) Inducing Lattices in Non-Lattice Linear Problems. In Proceedings of the 42nd International Symposium on Reliable Distributed Systems. SRDS 2023. (extended version of brief announcement [2])
- [3] Gupta, A. T. and Kulkarni, S. S. (2023) Lattice Linearity of Multiplication and Modulo. In: Dolev, S., Schieber, B. (eds) Stabilization, Safety, and Security of Distributed Systems. SSS 2023.
- [2] Gupta, A. T. and Kulkarni S. S. (2022) Brief Announcement: Fully Lattice Linear Algorithms. In: Devismes, S., Petit, F., Altisen, K., Di Luna, G.A., Fernandez Anta, A. (eds) Stabilization, Safety, and Security of Distributed Systems. SSS 2022. (full conference paper: [4])
- [1] Gupta, A. T., and Kulkarni, S. S. (2021) Extending Lattice Linearity for Self-stabilizing Algorithms. In: Johnen C., Schiller E.M., Schmid S. (eds) Stabilization, Safety, and Security of Distributed Systems. SSS 2021. (journal version: [5])