Distributed Network Algorithms

Gopal Pandurangan

Department of Computer Science
University of Houston
February 20, 2019
To
Sumithra, Iniya, and Sathyan
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>i</td>
</tr>
<tr>
<td>List of Figures</td>
<td>ix</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Two aspects of distributed computing</td>
<td>2</td>
</tr>
<tr>
<td>1.2 What this book is about?</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Background needed</td>
<td>5</td>
</tr>
<tr>
<td>1.4 Worked Exercises and Exercises</td>
<td>5</td>
</tr>
<tr>
<td>1.5 Advanced Material</td>
<td>5</td>
</tr>
<tr>
<td>1.6 References</td>
<td>6</td>
</tr>
<tr>
<td>2 Model</td>
<td>7</td>
</tr>
<tr>
<td>2.1 The Distributed Network Model</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Performance of distributed algorithms: Complexity measures</td>
<td>12</td>
</tr>
<tr>
<td>2.2.1 Time complexity</td>
<td>12</td>
</tr>
<tr>
<td>2.2.2 Message complexity</td>
<td>13</td>
</tr>
<tr>
<td>2.3 Exercises</td>
<td>13</td>
</tr>
<tr>
<td>3 Broadcast and Tree Algorithms</td>
<td>15</td>
</tr>
<tr>
<td>3.1 Broadcast</td>
<td>15</td>
</tr>
<tr>
<td>3.1.1 Flooding Algorithm</td>
<td>15</td>
</tr>
<tr>
<td>Analysis of Flooding</td>
<td>16</td>
</tr>
<tr>
<td>3.2 A Lower Bound for Broadcast</td>
<td>17</td>
</tr>
<tr>
<td>3.3 Tree Broadcast</td>
<td>18</td>
</tr>
<tr>
<td>3.4 Convergecast on a Tree</td>
<td>19</td>
</tr>
<tr>
<td>3.4.1 Pipelining</td>
<td>20</td>
</tr>
<tr>
<td>3.5 Upcast</td>
<td>20</td>
</tr>
<tr>
<td>3.6 Downcast</td>
<td>22</td>
</tr>
<tr>
<td>3.7 Constructing a BFS spanning tree</td>
<td>22</td>
</tr>
<tr>
<td>3.7.1 Distributed BFS tree construction algorithm</td>
<td>23</td>
</tr>
</tbody>
</table>
## CONTENTS

6.2 Coloring ................................................. 79
   6.2.1 A $(\Delta + 1)$-Coloring Algorithm ................. 79
6.3 Deterministic Symmetry Breaking .............................. 82
   6.3.1 Coloring in Rooted Trees .............................. 82
   From 6-coloring to 3-coloring ............................. 83
   6.3.2 $\Delta + 1$-coloring of bounded degree graphs ...... 84
6.4 Dominating set ........................................... 85
   6.4.1 Set Cover Problem .................................... 86
   Performance of Greedy Algorithm ......................... 87
   Distributing Greedy ....................................... 88
   6.4.2 Distributed Algorithm for Dominating Set .......... 89
   Analysis ................................................... 90
6.5 Worked Exercises ......................................... 92
6.6 Exercises ................................................ 94

7 Minimum Spanning Tree .................................... 101
   7.1 Gallager-Humblet-Spira (GHS) Algorithm ............... 102
      7.1.1 One phase of GHS .................................. 103
      Find MOE of all fragments ............................... 104
      Merging fragments via their MOEs ...................... 104
      7.1.2 Analysis of GHS algorithm ......................... 105
   7.2 The Pipeline Algorithm ................................ 106
      7.2.1 Analysis ........................................... 106
   7.3 Garay-Kutten-Peleg (GKP) Algorithm: An $O(D + \sqrt{\pi \log^* n})$-round algorithm ...................... 109
      7.3.1 First part: Controlled-GHS algorithm .............. 110
      7.3.2 Second Part: Combining the Remaining $\sqrt{\pi}$ fragments ......... 114
      7.3.3 The Overall Algorithm ............................. 115
   7.4 Worked Exercises ....................................... 115
   7.5 Exercises ............................................... 116

8 MapReduce Algorithms .................................. 119
   8.1 Overview of MapReduce ................................ 119
      8.1.1 Example: Counting Word Frequencies ............. 120
   8.2 A Model of Computation for MapReduce ................ 121
   8.3 MapReduce Algorithms ................................. 122
      8.3.1 Densest Subgraph ................................ 122
      8.3.2 Minimum Spanning Tree (MST) .................... 125
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td><em>k</em>-machine Model and Algorithms</td>
<td>129</td>
</tr>
<tr>
<td>9.1</td>
<td>Introduction</td>
<td>129</td>
</tr>
<tr>
<td>9.2</td>
<td>Model</td>
<td>131</td>
</tr>
<tr>
<td>9.3</td>
<td>Designing algorithms for the <em>k</em>-machine model</td>
<td>133</td>
</tr>
<tr>
<td>9.3.1</td>
<td>A Mapping Lemma</td>
<td>134</td>
</tr>
<tr>
<td>9.3.2</td>
<td>The Conversion Theorem</td>
<td>136</td>
</tr>
<tr>
<td>9.4</td>
<td>Algorithms for some problems</td>
<td>140</td>
</tr>
<tr>
<td>9.4.1</td>
<td>Breadth-First Search Tree (BFS)</td>
<td>140</td>
</tr>
<tr>
<td>9.4.2</td>
<td>Minimum Spanning Tree (MST), Spanning Tree Verification (ST) and Graph Connectivity (Conn)</td>
<td>140</td>
</tr>
<tr>
<td>9.4.3</td>
<td>PageRank</td>
<td>141</td>
</tr>
<tr>
<td>9.5</td>
<td>Exercises</td>
<td>142</td>
</tr>
<tr>
<td>10</td>
<td>Basics of Asynchronous Systems</td>
<td>143</td>
</tr>
<tr>
<td>10.1</td>
<td>Measuring Time in an Asynchronous System</td>
<td>143</td>
</tr>
<tr>
<td>10.1.1</td>
<td>Logical Time and the Happens Before Relation</td>
<td>144</td>
</tr>
<tr>
<td>10.1.2</td>
<td>Vector Clocks</td>
<td>145</td>
</tr>
<tr>
<td>10.1.3</td>
<td>Consistent Cuts</td>
<td>148</td>
</tr>
<tr>
<td>Appendices</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Appendix A</td>
<td>Asymptotic Notation</td>
<td>153</td>
</tr>
<tr>
<td>A.1</td>
<td>Big-O notation</td>
<td>153</td>
</tr>
<tr>
<td>A.2</td>
<td>Big-Ω notation</td>
<td>154</td>
</tr>
<tr>
<td>A.3</td>
<td>Big-Theta (Θ) Notation</td>
<td>155</td>
</tr>
<tr>
<td>A.4</td>
<td>Little-o and Little-ω notations</td>
<td>155</td>
</tr>
<tr>
<td>A.5</td>
<td>Examples</td>
<td>156</td>
</tr>
<tr>
<td>Appendix B</td>
<td>Mathematical Induction</td>
<td>159</td>
</tr>
<tr>
<td>B.1</td>
<td>The General Technique</td>
<td>159</td>
</tr>
<tr>
<td>B.1.1</td>
<td>Sum of first <em>n</em> natural numbers</td>
<td>160</td>
</tr>
<tr>
<td>B.1.2</td>
<td>Coloring regions formed by lines</td>
<td>161</td>
</tr>
<tr>
<td>B.2</td>
<td>Strong Mathematical Induction</td>
<td>162</td>
</tr>
<tr>
<td>B.2.1</td>
<td>Fundamental Theorem of Arithmetic</td>
<td>163</td>
</tr>
<tr>
<td>B.3</td>
<td>Using Induction for Algorithm Analysis</td>
<td>163</td>
</tr>
<tr>
<td>B.4</td>
<td>Worked Exercises</td>
<td>164</td>
</tr>
<tr>
<td>B.5</td>
<td>Exercises</td>
<td>165</td>
</tr>
</tbody>
</table>
Appendix C  Probability  

C.1  Events, Probability, Probability Space  

C.2  Principle of Inclusion-Exclusion  

C.3  Conditional Probability  

C.4  The Birthday Paradox  

C.5  Random Variable  

C.6  Expectation of a random variable  

C.7  Useful Types of Random Variables  

C.7.1  Binomial Random variables  

C.7.2  The Geometric Distribution  

C.8  Bounding Deviation of a Random Variable from its Expectation∗  

C.8.1  Markov Inequality  

C.8.2  Chernoff Bound for Sum of Indicator R.V's  

C.9  Moment Generating Function∗  

C.9.1  Proof of Chernoff Bound for Sum of Indicator R.V's  

C.9.2  Example application of Chernoff bound  

C.10  Conditional Expectation  

Appendix D  Graph-theoretic Concepts  

D.1  Definition  

D.2  Path, Cycle, and Tree  

D.3  Distance  

D.4  Subgraph and Induced Subgraph  

D.5  Degree of a vertex  

D.6  Connectivity  

D.7  Spanning Tree (ST)  

D.7.1  The Size of a Spanning Tree  

D.8  Graph Search Algorithms  

D.8.1  Breadth-first search (BFS)  

D.8.2  Depth-first search (DFS)  

Appendix E  Commonly Used Math Formulas in Algorithm Analysis  

E.1  Geometric series  

E.2  Combinatorial Inequalities  

Bibliography
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A distributed network modeled as a graph.</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Distributed computing model classification</td>
<td>11</td>
</tr>
<tr>
<td>3.1</td>
<td>Illustrating the Flooding algorithm in a graph. $s$ is the source node.</td>
<td>16</td>
</tr>
<tr>
<td>3.2</td>
<td>Illustrating the Convergecast algorithm in a graph. $s$ is the source node.</td>
<td>19</td>
</tr>
<tr>
<td>3.3</td>
<td>Illustrating Upcast Algorithm in a graph. Ties are broken by priority based on ID.</td>
<td>21</td>
</tr>
<tr>
<td>3.4</td>
<td>BFS tree construction algorithm on a graph starting from source node $S$. Figures (1)-(6) show the steps of the algorithm. The BFS tree constructed is shown. The table shows the parent-child relationships and the invite-accept steps executed.</td>
<td>24</td>
</tr>
<tr>
<td>3.5</td>
<td>Illustrating termination detection. Steps (1)-(4) show the echo sending (convergecast) process. Once the echos reach the root, the source node broadcasts “done” message to all the nodes (via flooding). The broadcasting is shown in steps (5)-(7).</td>
<td>25</td>
</tr>
<tr>
<td>4.1</td>
<td>The figure shows routing tables in each node in a graph. (If slot is empty in a routing table, then it means it is a neighbor.) For example, if node $B$ wants to route a message to node $E$, then according to $B$’s routing table the next hop is node $D$ and so it will forward the message to $D$ (along with the information of the destination node, which is $E$). According to $D$’s routing table, $E$ is a neighbor, so it will forward to $E$ directly.</td>
<td>39</td>
</tr>
<tr>
<td>4.2</td>
<td>Illustrating Dijkstra’s algorithm (centralized) on a graph. The algorithm computes shortest paths from $A$ to all nodes in the graph; the output can be represented as a shortest path tree (SPT). The final SPT is shown — the (unique) path in the tree from $A$ to any other node gives the shortest path to that node.</td>
<td>40</td>
</tr>
</tbody>
</table>
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3</td>
<td>Illustrating Belman-Ford distributed algorithm on a graph. The algorithm computes shortest paths from A to all nodes in the graph; the output can be represented as a shortest path tree (SPT). The final SPT is shown — the (unique) path in the tree from A to any other node gives the shortest path to that node.</td>
<td>41</td>
</tr>
<tr>
<td>4.4</td>
<td>An example network for count-to-infinity problem.</td>
<td>42</td>
</tr>
<tr>
<td>5.1</td>
<td>Illustrating BasicLE-Ring algorithm.</td>
<td>64</td>
</tr>
<tr>
<td>5.2</td>
<td>Symmetry breaking using randomization.</td>
<td>65</td>
</tr>
<tr>
<td>5.3</td>
<td>Illustrating RandLE-Ring algorithm.</td>
<td>66</td>
</tr>
<tr>
<td>5.4</td>
<td>Illustrating RandLE-CN algorithm.</td>
<td>67</td>
</tr>
<tr>
<td>5.5</td>
<td>Illustrating BasicLE-Gen algorithm.</td>
<td>68</td>
</tr>
<tr>
<td>6.1</td>
<td>Maximal Independent Set.</td>
<td>71</td>
</tr>
<tr>
<td>6.2</td>
<td>Illustrating MIS Algorithm 1</td>
<td>98</td>
</tr>
<tr>
<td>6.3</td>
<td>Illustration for proof of Lemma 6.1.</td>
<td>99</td>
</tr>
<tr>
<td>7.1</td>
<td>GHS algorithm: Finding MOE of all fragments in parallel.</td>
<td>103</td>
</tr>
<tr>
<td>7.2</td>
<td>GHS algorithm: Merging fragments.</td>
<td>104</td>
</tr>
<tr>
<td>7.3</td>
<td>Controlled-GHS algorithm: Merging fragments. The figure shows a fragment graph consisting of 8 fragments — a, b, c, d, e, f, g, h. In the fragment graph each fragment is represented as a node and the edges between fragments are the MOE edges — the outgoing edges are directed away from the respective fragments. The red edges are the edges of a maximal matching. The fragments are merged via the matched edges plus the added green edges; a green edge is the MOE edge of a fragment which has no incident matching edge — in the above example, these are fragments a and c. The figure shows the merged fragments after merging via the red and green edges.</td>
<td>118</td>
</tr>
<tr>
<td>10.1</td>
<td>A Space-Time Diagram (left) and the corresponding Space-Time Graph (right).</td>
<td>144</td>
</tr>
<tr>
<td>10.2</td>
<td>A run of the logical clocks algorithm.</td>
<td>145</td>
</tr>
<tr>
<td>10.3</td>
<td>A run of an asynchronous algorithm where each step is labeled with a vector clock reading.</td>
<td>147</td>
</tr>
<tr>
<td>10.4</td>
<td>Two different space-time visualizations of the same asynchronous computation. Note that the consistent cuts C and D represent equivalent global snapshots, whereas E is an inconsistent cut.</td>
<td>149</td>
</tr>
</tbody>
</table>
A.1 Illustration of the Big-O notation. In both figures, \( g(n) = O(f(n)) \).

Left: \( f(n) \geq g(n) \) for all \( n \geq n_0 \), where \( n_0 \) is a fixed constant.

Note that what happens when \( n < n_0 \) is irrelevant. Right:

\( g(n) \geq f(n) \), but \( g(n) \leq 5f(n) \), for all \( n \). .......................... 154

D.1 A connected undirected graph (left) and the same graph with weights (right). .......................... 184

D.2 A graph and an induced subgraph of the graph. ....................... 185

D.3 A disconnected graph. ........................................... 186

D.4 An undirected graph and its BFS tree. ................................. 187

D.5 An undirected graph and its DFS tree rooted at vertex \( a \) where the DFS is started. ................................. 188
Chapter 1

Introduction

This book will cover the fundamentals of distributed network algorithms. With the rise of the Internet and the paradigm shift from single-processor computing to multi-processor computing, it becomes very important to understand the fundamentals of distributed computing and networks. The examples of real-world distributed networks are many: the Internet consists of billions of entities connected over a communication backbone; a peer-to-peer network consists of computers communicating via an overlay network built on top of the Internet; wireless phones can communicate with each other or through a centralized access point using radio communication; a sensor network consists of tiny devices that communicate using a wireless medium; a typical data center network consists of thousands of distributed machines connected over a network which can be used to distributively solve a large-scale computational problem; Bitcoin is a peer-to-peer digital currency that is managed in a decentralized fashion; a multicore computer consists of many processors that communicate via a shared memory; a network of people communicating via a social network. In all these examples, we have the paradigm of many distributed agents that communicate via some network or shared medium to jointly perform some task. The study of computation of such distributed systems is quite different from the traditional computation theory of single processor (sequential) computation (which you study in a typical algorithms course). While time and memory are the important complexity measures in the sequential setting, the amount of communication between nodes plays a key role in the distributed setting.

This book introduces the fundamentals of distributed computing focusing on models, algorithms, and complexity. Various fundamental distributed network algorithms including broadcast, convergecast, maximal independent set, coloring, leader election, spanning tree algorithms, shortest paths, and routing will
be covered. Basic issues arising in distributed systems such as communication, synchronization, fault-tolerance, and agreement will be addressed. Applications to real-world networks such as the Internet, peer-to-peer networks, wireless networks, sensor networks and dynamic networks will be discussed.

1.1 Two aspects of distributed computing

Traditionally distributed algorithms have been taught in the context of distributed systems. This is, of course, the main setting for studying such algorithms as they underlie the operation of distributed systems and communication networks. Hence, this will be one of the main aspects that we will address in this book. However, we will also address another aspect, which has received traditionally much less attention, but has grown in importance over the years. This aspect uses distributed computing for doing computation — i.e., either speeding up computation (in the sense of high performance or parallel computing) or distributed processing of large-scale data or both. Instead of using dedicated parallel computers (such as super computers), the goal is to use several (may be thousands) of computers distributed over a network to solve computational/algorithmic problems. In particular, we will study how distributed algorithms can be used to solve large-scale graph problems, where the graphs (such as the Web graph) can be so large that it may not even fit in one machine and hence has to be distributed across the machines. Here the goal is using distributed algorithms to solve problems efficiently on the input graph that is distributed across the machines. While the two aspects have different goals, we will see that they are closely related; in fact the algorithms and techniques developed under the first aspect find ready applicability in tackling problems in the second aspect.

1.2 What this book is about?

This book will focus mainly on the algorithmic and theoretical underpinnings of distributed computing. In particular, fundamental algorithms in distributed computing will be covered. Distributed computing has blossomed into a large field with various computational models and numerous associated algorithmic results. However, the study of the field can be categorized using a few important modeling criteria. The first and foremost is the communication mechanism which is essentially of two types: message passing and shared memory.

Message passing vs. shared memory. In message passing, nodes (we will use this terminology throughout which can denote machines, processors, sensors,
1.2. WHAT THIS BOOK IS ABOUT?

people, or in general, any set of agents) communicate via an underlying network by exchanging messages. Whereas in shared memory, the communication is via shared access to a common memory. While some fundamental problems in distributed computing are common to both models, some others are more meaningful in one versus the other. In this book we will focus on the message passing model. In a message passing model, we assume an underlying communication network modeled as a graph. The vertices (or interchangeably nodes) of the graph represent the nodes and the edges of the graph represent the communication links between the nodes. Two neighboring nodes are nodes that are connected to each other directly via a communication link; such nodes can communicate with each other by sending and receiving messages.

**Synchrony vs. asynchrony.** A second important criterion is timing: synchrony versus asynchrony. In a synchronous system, the actions of nodes are fully synchronized and they act in lock-step. There are no delays in communication, and all processors perform computation in synchrony. A synchronous system has the notion of timesteps partitioned into discrete rounds: in each round, each processor typically does a computation and a send/receive operation — the message sent to a neighbor is assumed to be at the beginning of the round and is received by the neighbor at the end of the same round with no delay. A synchronous system can also be thought of operating using a common global clock; all nodes have access to this clock and each clock tick (or a certain number of clock ticks) correspond to a round. Alternatively, one can think of each node having its own local clock which are all synchronized. Synchronous systems do not strictly exist in practice and are only an approximation to real-world distributed systems. Real-world distributed systems are much more complicated. In real systems, clocks are typically not synchronized (even if they are synchronized at the "beginning", they will eventually start drifting), there is an unpredictable communication delay incurred for messages, and processors can operate at different speeds. However, as we will see later, the synchronous model is very useful for designing distributed algorithms; more importantly, the proof of correctness is typically simpler to show; furthermore lower bounds proved for this model, typically apply for the asynchronous model as well. Most of the algorithms in this course will use the synchronous model (unless otherwise stated).

On the other hand, in an asynchronous system, there is no access to a global clock, and there can be arbitrary communication delays in sending and receiving messages. There is no notion of a common round. In particular, a message can incur an arbitrary, but finite delay. Each node may have access to a local clock,
which are not necessarily synchronized; thus a notion of a common time does not exist. One has to define alternate notions of time in an asynchronous system. Asynchronous systems represent “worst-case” behavior of real-world systems (assuming no failures — we will come to this aspect later) and hence algorithms designed to run correctly on asynchronous systems will work correctly for real-world distributed systems. This is a big advantage and motivation to work with the asynchronous model. However, proving correctness of algorithms, a key criterion, can be much more complicated, thus making the life of the distributed algorithm designer hard.

Efficient algorithms. Another aspect that will be stressed throughout is the efficiency of distributed algorithms. The efficiency measures that we consider are time and message complexity. Designing algorithms that finishes as quickly as possible and uses as little communication (messages) as possible will be an important focus. We will also cover some lower bounds that show that certain problems cannot be solved faster or using less communication (messages). These are important as well, as it shows the limits of performance that a distributed algorithm designer can achieve.

Failures and faults. An important aspect of distributed computing is fault-tolerance; faults, failures, and even malicious behavior of nodes (and links as well) is generally a norm rather than an exception in distributed computing. A famous saying of Leslie Lamport who won a Turing award for his contributions to distributed computing is: “A distributed system is one in which the failure of a computer you didn’t even know existed can render your own computer unusable.”

While the course will focus mostly on algorithms in the fault-free setting, we will also study distributed algorithms that operate under faults. In the faulty setting, the differences between synchronous and asynchronous models become even more pronounced and it turns out that some fundamental problems cannot be solved in the asynchronous setting under the presence of faults.

Real-world distributed networks. The final part of the course will focus on real-world distributed networks: peer-to-peer networks, wireless and sensor networks. We will also look at distributed models for computing with large-scale data characterized by systems such as MapReduce and Pregel.

What will you learn? The book aims to teach the fundamentals of distributed network algorithms. It is expected that you will acquire a good knowledge of distributed algorithms, in particular, algorithms for various fundamental
problems, which form building blocks for designing efficient algorithms in distributed networks and systems. You will also appreciate the limits to the efficiency of distributed algorithms, including what is possible and what is not. This will give insight into designing algorithms that can operate close to the theoretical optimum. From a practical point of view, as large-scale distributed systems and networks become increasingly common, these algorithms will play an important role in designing efficient fault-tolerant distributed systems.

1.3 Background needed

For this course, it is important to have a good background in discrete mathematics and algorithms, at least at the undergraduate level. You will find some background material in the appendix of this book on mathematical basics of algorithm analysis, such as asymptotic analysis and mathematical induction. It is highly recommended that you refresh your background by reviewing a textbook in Algorithms. For example, you can see the algorithms book available under my webpage at [https://sites.google.com/site/gopalpandurangan/home/algorithms-course](https://sites.google.com/site/gopalpandurangan/home/algorithms-course).

You also have a good programming experience in at least one high-level programming language, e.g., C, C++, Pascal, Java, Python, Lisp, ML etc.

1.4 Worked Exercises and Exercises

Each chapter has numerous exercises which complement the material covered in the chapters. Solving the exercises is a good exposure to solving algorithmic problems. Each chapter also has several “worked exercises”; these are important in further understanding and applying the concepts covered in the respective chapters. Trying to solve the worked exercises before seeing their solutions is a good learning technique. The solutions to the worked exercises can provide insights in solving the exercises.

1.5 Advanced Material

Headings of some chapters and sections of the book are suffixed by stars (*). These are (somewhat) advanced material and can be skipped during first reading. Exercises and Worked Exercises with stars are problems with above average difficulty.
1.6 References

The primary reference will be this book. There are a lot of excellent textbooks in distributed algorithms which give a different perspective or cover material that is not covered here. The following list is not exhaustive.

- Distributed Computing: A Locality Sensitive Approach by David Peleg.
- Distributed Algorithms by Nancy Lynch.
- Introduction to Distributed Algorithms by Gerard Tel.
- Distributed Algorithms by Wan Fokkink.
- Distributed Computing Notes by Roger Wattenhofer at http://www.dcg.ethz.ch/lectures/podc/
Chapter 2

Model

The focus of this course is on distributed network algorithms which are algorithms that are designed to run on many processors “distributed” over a communication network. Typically, the processors are distributed over a geographical area. Examples include Internet, peer-to-peer networks, ad hoc wireless and sensor networks. As mentioned in the previous chapter, there are many distributed computing models. This course will focus almost exclusively on the message passing model which is suitable for distributed computation by a set of processors interconnected over a communication network.

2.1 The Distributed Network Model

The following are the key ingredients of the distributed computing model that we will use.

Connected undirected graph A distributed network is modeled as a connected, undirected graph. Nodes are processors and edges are communication links. Since we assume an undirected graph, communication is possible in both ways, i.e., an undirected edge \((u, v)\) allows message passing from \(u\) to \(v\) as well as \(v\) to \(u\). Sometimes, edges can be directed, i.e., only one way communication is allowed. However, unless otherwise stated, the graph will be undirected. We will typically use the notation \(G = (V, E)\) to denote a graph or a network. \(V\) stands for the set of nodes and \(E\) for the set of edges. Unless otherwise stated, the size of the network, i.e., the number of nodes in \(G\), is denoted by \(n = |V|\), and the number of the edges is denoted by \(m = |E|\).

Sometimes, we will use a weighted graph to model a network, i.e., edges
will have an associated cost function (assumed to be rational-valued): \( c \); 
\( c(e) \) denotes the cost (or weight) of edge \( e \), where \( c(e) \) is a rational number. Costs are typically positive, but sometimes can be negative as well. Costs can model various parameters associated with edges, e.g., bandwidth, delay, distance etc. By default, we will consider a graph to be unweighted. An unweighted graph can be considered as a special case of a weighted graph where all costs are 1.

Figure 2.1 gives an example of a connected undirected graph.

Figure 2.1: A distributed network modeled as a graph.

**Local communication** Nodes can communicate directly (only) with their neighbors through the edges. We can distinguish two types of local communication: (1) *Local unicast*: nodes can send different messages to each of its neighbors in one step; (2) *Local broadcast*: nodes send the same message to all its neighbors in any step. Local broadcast is simpler and is a feature of wireless networks, which use radio broadcast. On the other hand, local unicast, is more suitable to wired networks.

**Unique IDs** Nodes have unique processor identities (IDs). For example, in the Internet, each endhost (may be an intermediate router or a processor at an endpoint) has a unique IP address. There are situations where node ids may not be present or they may not be unique; for most part we will not consider this scenario, although such anonymous networks (or networks with homonyms) are also worth studying.

**Synchrony** Two important models can be distinguished based on processor synchronization.
2.1. THE DISTRIBUTED NETWORK MODEL

Synchronous model: In a synchronous model, each processor has an internal clock and the clocks of all processors are synchronized. We assume that the processor speeds are uniform and each processor takes the same amount of time to perform the same operation. Computation proceeds in lock-step in a series of discrete rounds (time steps). In each round, each processor (node) can do some local (internal) computation and can also send/receive messages. To be concrete, we assume that at the beginning of a round, a node receives messages (if any) from its neighbors via its incident edges. It does some local computation based on these received messages and on its internal state; this can lead to a change of its internal state as well as sending messages (if any) to some or all of its neighbors at the end of the round. Thus a round consists of receive-compute-send in that order. The messages sent by a node is received by its neighbor at the beginning of the next round. A node can send messages simultaneously through all its edges at once in a round. Unless otherwise stated, we will use the synchronous model.

Asynchronous model: No assumptions are made about any internal clocks or on the speeds of the processors. The steps in an asynchronous algorithms are determined by an event schedule and not by clock ticks. However, we do make two reasonable timing assumptions. First, we assume that messages arrive in the same order they are sent (i.e., there is FIFO queuing). Second, we assume that if a processor has an event enabling it to perform a task, the processor will eventually perform the task. In particular, each message sent will eventually be delivered.

Between the two extreme models, we can define “intermediate” models that are partially synchronous, where the processors have some partial information about timing (e.g., almost synchronized clocks or upper bounds on message delivery time etc.), but do not have complete information as they do in the synchronous model. Although intermediate models can provide a more realistic model of real networks such as the Internet, we will restrict our attention to synchronous and asynchronous models in this course. Algorithms designed for the synchronous model, can sometimes be translated to work for the asynchronous model (see below), and algorithms for the latter model will work for an intermediate model as well.

Local knowledge Local knowledge is an important issue in distributed algorithms. We assume that nodes initially, i.e., at the beginning of computation, have only limited knowledge: each node knows only about itself
and its incident edges; it may not know anything about the identities of its neighbours or their internal states. One way of thinking about this is that, each node has a port associated with an incident edge, with each port having a port number. Edges connect ports of two neighbouring nodes and each node only knows about its own port number and that an edge goes out of it, but nothing about the other endpoint of the edge. This model is referred to as the clean network model and is also sometimes referred to as the KT0 model, i.e., the initial (K)nowledge of all nodes is restricted (T)ill radius 0 (i.e., just the local knowledge). The KT0 model is a standard model in distributed computing and typically used. Unlike the KT0 model, one can assume a model where nodes have initial knowledge of their neighbors, especially their IDs. This model is called KT1 model. (One can generalize this and assume knowledge up to radius \(r\), i.e., each node has knowledge all nodes within distance \(r\) from it). This model is used relatively infrequently; we will revisit this model later. Unless otherwise stated, throughout we will assume the clean network (KT0) model.

Sometimes, we will assume, that nodes have some limited global knowledge, e.g., about the network size (the number of nodes in the network) — denoted throughout by \(n\). A node has to communicate with its neighbors in order to know about their identities; throughout the run of the algorithm, it can also learn about the neighbors of its neighbors, by communicating with them (indirectly via its neighbours) etc.

**CONGEST vs. LOCAL.** Depending on whether there is a restriction on the size of message that can be sent through an edge in a round, we distinguish two important models: CONGEST and LOCAL. In the CONGEST model, the size (measured in bits) of each message is small, typically of size \(O(\log n)\), i.e., only a message of size at most \(O(\log n)\) bits can be sent per edge per round, where \(n\) is the size of the network (the total number of nodes). This bound is reasonable since at least \(\Theta(\log n)\) bits are needed to uniquely address a node; thus this allows a message to contain the ID of a node. In many algorithms, this might be needed to identify the sender and/or the receiver. In the LOCAL model, there is no restriction on the size of a message; message sizes can be arbitrary large. The CONGEST model captures the inherent bandwidth restriction that is present in real-world networks. However, as we shall see, the LOCAL model is useful in focusing on locality issues in distributed computing.

**Operation of a distributed algorithm** How do distributed algorithms operate? We will assume that each node will start executing its instance of the algo-
2.1. THE DISTRIBUTED NETWORK MODEL

algorithm that is given to it at the beginning of the computation. Throughout this book, unless otherwise stated, every node will operate the same instance of the algorithm. However, depending on its local information, each node can have its own behaviour. The most common scenarios for this are: (1) Unique IDs: since each node has its own unique ID, this can influence how each node behaves and (2) Randomization: Each node might make its own random choices which can influence the future course of the algorithm; however, the specification of the algorithm (i.e. its “source code”) will be same for all nodes. We will discuss randomized algorithms later.

Starting time How and when does a node start its execution? We will typically assume that all nodes start the computation at round 0, are initially in the same starting state, and proceed to execute the same algorithm. However, in some algorithms, only some nodes start their computation at round 0; other nodes can join later in the computation when they receive messages from their neighbors. These issues will become clearer when we discuss various algorithms.

For most part, we will assume the synchronous CONGEST model unless otherwise stated, since it is simpler and easier to design algorithms for this model. Another key plus of this model is that it allows us to focus on certain core distributed algorithmic issues, without the distraction of more complex issues that arise in the asynchronous setting. In the fault-free setting, we will see later that using a tool called synchronizers one can transform a synchronous algorithm to work in an asynchronous model without increasing the overall complexity too much.
However, this does not mean that asynchronous model or the LOCAL model is less important. We will study these models when and where they are appropriate. For example, to understand locality issues, i.e., how far (i.e., how many hops) a node must communicate to achieve a certain task, it might be more appropriate to adopt the LOCAL model, which abstracts out issues related to bandwidth and congestion. Another important issue that we will ignore for now is failures and faults. Unless otherwise stated, we will assume that nodes do not fail and are not malicious, i.e., they faithfully run the algorithm they are supposed to execute. Moreover, messages are delivered correctly and are not lost or otherwise corrupted. Later in the course, we will study models with failures whenever appropriate.

A relevant question is whether the above synchronous model is too restrictive? One can think of the model as mainly to facilitate analysis — it provides a clean way to compare the performance of various distributed algorithms (this situation is similar to the RAM model in the sequential setting[3]). Algorithms themselves need not be conformed to work in this restricted model; they may work in more general or alternate models. The synchronous/asyncronous models are the two extremes. If we have an algorithm that assumes point-to-point communication (suitable for a wired setting), can we use it in a wireless setting where radio broadcast communication is used? Some elements will have to be changed, but the core algorithm may still work.

2.2 Performance of distributed algorithms: Complexity measures

There are two important complexity measures for the performance of distributed algorithms: time complexity and message complexity.

2.2.1 Time complexity

The first is the time complexity, or the time needed for the distributed algorithm to terminate to a solution. In the synchronous model, time is measured by the number of clock ticks called rounds, i.e., processors are said to compute in “lock step”. When running a distributed algorithm, different nodes might take a different number of rounds to finish. In that case, the maximum time needed over all nodes is taken as the time complexity.

For an asynchronous algorithm, this definition is meaningless, since a single message from a node to a neighbor can take a long time to arrive. Therefore we use the following definition instead: Time complexity in an asynchronous model
2.3. **EXERCISES**

is the time units from start to finish, assuming that each message incurs a delay of **at most one time unit**. Note that this definition is used only for **performance analysis** and does not imply that there is a bound on delay in asynchronous networks.

### 2.2.2 Message complexity

The second important measure is **message complexity**, which measures the total number of messages that are sent between all pairs of nodes during the computation. Recall that, in the CONGEST model, each message has small size, i.e., is assumed to be of $O(\log n)$ bits. In the LOCAL model, messages of large sizes can be sent and so the number of messages might not give us an accurate picture of the communication cost. In such situations, the **bit complexity**, i.e., the number of bits transmitted per edge per round is a better measure.

### 2.3 Exercises

**Exercise 2.1.** Show that $\Theta(\log n)$ bits are both necessary and sufficient to uniquely address a network of $n$ nodes, when each node has a (unique) identifier chosen from the integer range $[1, n^2]$.

**Exercise 2.2.** Consider a distributed network of $n$ nodes, where each node starts with its own data value, say a number. For example, this can be a sensor network and the value can represent the temperature reading of each sensor node. Describe a distributed algorithm that computes the minimum value among all nodes of the network. When the algorithm has terminated, every node should know the value of the minimum. You can assume the synchronous CONGEST model.

Show that your algorithm correctly computes the global minimum. Does your algorithm detect termination, i.e., does each node know when all nodes have finished their computation? What is the time and message complexity of the algorithm? Does your algorithm and analysis also work in the asynchronous model?

**Exercise 2.3.**

1. What is the difference between the KT0 model and the KT1 model?

2. Suppose we have an algorithm that works in the KT0 model. Will it work in the KT1 model without any changes?

3. Suppose we have an algorithm that works in the KT1 model. Will it work in the KT0 model without any changes?
4. Suppose we show that the time complexity of solving some problem in the KT1 model is $T$, then what does it imply regarding the time complexity of the problem in the KT0 model?

5. Answer the above question if “time complexity” is replaced by “message complexity”.

Chapter 3

Broadcast and Tree Algorithms

We start our study of distributed algorithms with some of the most basic algorithms. These basic “primitives” are commonly used and they also form building blocks for more sophisticated algorithms. As discussed in Chapter 2, we will assume that the distributed network is an undirected connected graph. We will consider the CONGEST model, i.e., only $O(\log n)$ sized message can be sent per edge per time step. Our algorithms will also assume the synchronous model throughout.

This chapter uses basic concepts in graph theory; for a quick recap of these, see Appendix D.

3.1 Broadcast

Broadcasting is an important communication mode: sending a message from a source node to all other nodes of the network.

**Definition 3.1.1** (Broadcast). Given a network $G = (V, E)$ and a source node $s$, the goal of broadcast is to send a message from $s$ to all nodes in $V$.

Two basic broadcasting approaches are flooding and spanning tree-based.

3.1.1 Flooding Algorithm

A source node $s$ wants to send a message $M$ to all nodes in the network. $s$ simply forwards $M$ over all its edges. Every vertex $v \neq s$, upon receiving $M$ for the first time (over an edge $e$) forwards (sends) it on every other edge. Upon receiving the message again $v$ does nothing. Algorithm 3.1 gives the pseudocode of the
Flooding algorithm and Figure 3.1 illustrates it. We note that the pseudocode is written from the perspective of a single node \( v \). All nodes will execute the same algorithm — this will be case for all algorithms considered in this course, unless otherwise stated.

**Algorithm 1** Flooding Algorithm; code executed by node \( v \).

1: \textbf{if } \( v = s \) \textbf{then}  
2: \hspace{1em} send message \( M \) to all neighbours and stop  
3: \textbf{else if } \( M \) is received for the first time \textbf{then}  
4: \hspace{1em} send \( M \) to all neighbors and stop

![Figure 3.1: Illustrating the Flooding algorithm in a graph. \( s \) is the source node.](image)

**Analysis of Flooding**

Flooding is an easy algorithm to analyze. Analysis of distributed algorithms involves showing two things: correctness and complexity bounds. Correctness involves showing that the algorithm does indeed accomplish the expected goal. Complexity bounds involves bounding the amount of time and the number of messages consumed by the algorithm.

**Theorem 3.1.** The Flooding algorithm is correct, i.e., all nodes eventually receive the message from the source node. The message complexity of Flooding is \( \Theta(|E|) \) and the time complexity is \( D \), where \( D \) is the diameter of \( G \).

**Proof Idea:** It is important to first get an intuition why the algorithm works before one formally shows it. In the case of Flooding, it is clear that every node
3.2. A LOWER BOUND FOR BROADCAST

will eventually receive the message sent by $s$. Why? Consider any node $v$. There is a path from $s$ to $v$, in particular, consider a shortest path between $s$ and $v$. Each node along this path forwards the message to its neighbour on the path and eventually it reaches $v$. Clearly the number of steps taken is bounded by the number of edges in the path, which is bounded by diameter $D$ of the network. Recall that the diameter is the maximum distance between any two nodes in the network.

Typically, we will use mathematical induction to argue for formal correctness of many algorithms. For a quick recap of induction, see Appendix B.

Proof. The formal correctness as well as time needed can be easily established via mathematical induction. As in the proof idea, let $v$ be any node. We use induction on $t$ to show that after $t$ time units, the message has already reached every vertex at a distance of $t$ or less from the source. At time $t = 0$ (the base case), node $s$ has the message, which is at distance 0. Assuming that the hypothesis is true at time $t$, it follows that at time $t + 1$, all neighbors of nodes at distance $t + 1$ from the source will receive the message by the flooding algorithm.

The message complexity follows from the fact that each edge delivers the message at least once and at most twice (one in each direction). The time complexity is clearly bounded by $D$, as discussed in the proof idea. 

3.2 A Lower Bound for Broadcast

Lower bounds are important to understand in algorithms; it tells us the (theoretical) optimum that can be attained by the performance of any algorithm. In this sense, lower bounds can be considered somewhat more “fundamental” and can be a useful guide in designing efficient algorithms.

In distributed algorithms, we can talk about lower bounds with respect to two measures: messages and time. More precisely, we talk about lower bounds for problems. For the broadcast problem, one can state the following “obvious” lower bound:

Theorem 3.2 (Lower Bounds for Broadcast). Any distributed algorithm for broadcast has a message complexity of $\Omega(n)$ and time complexity of $\Omega(D)$, where $n$ is the network size and $D$ is the network diameter.

Proof. Every node has to receive the message sent by $s$, the source node. Since there are $n$ nodes in the network, and each node (except $s$) has to receive a message on its own, at least $n - 1$ messages are needed.

For the time lower bound, note that, for any source node $s$, there exists a node (say $u$) which is at a distance of at least $\Omega(D)$ from $s$ (Exercise 3.1). Since
each round can advance the distance (to \( u \)) by at most one, the time lower bound follows.

Note that while the time lower bound of \( \Omega(D) \) is tight for broadcast (the flooding algorithm achieves it), the message lower bound is not. Later we will show a fundamental result: \( \Omega(m) \) is indeed a lower bound on the message complexity of broadcast.

### 3.3 Tree Broadcast

For tree broadcast, we need a spanning tree of the network.

**Definition 3.3.1** (Spanning Tree). A spanning tree \( T \) of a connected graph \( G \) is a subgraph of \( G \) that is a tree and spans all the vertices of \( G \). A spanning tree of \( G \) with \( n \) vertices has \( n - 1 \) edges.

Assume that we are given a spanning tree \( T \) of \( G \), rooted at \( s \). Then such a tree can be used to do efficient broadcast from \( s \) that can use less number of messages than flooding.

A spanning tree-based broadcast from a source node \( s \) proceeds as follows. The root \( s \) sends the message to its neighbours; when a node \( v \) receives the message for the first time, it forwards it along the tree edges only except to the node from which it just received the message; if \( v \) receives the message again, it ignores it.

Essentially, tree-based broadcast is like flooding, but restricted to the tree edges only; no messages are sent via non-tree edges.

The following theorem is immediate from the above description.

**Theorem 3.3.** Given a \( n \)-vertex graph \( G \) with a spanning tree \( T \) rooted at \( s \), the message complexity of broadcast from \( s \) is \( n - 1 \) and time complexity is \( \text{depth}(T) \), where \( \text{depth}(T) \) is the depth (or height) of the tree \( T \).

Note that the time complexity of spanning tree-based broadcast can be larger than the diameter of \( G \), if the depth of the spanning tree used is larger than diameter. If we use a breadth-first spanning tree (BFS) then, the depth of such a tree is diameter. Thus, using a breadth-first spanning tree, we get a message complexity of \( n - 1 \) and a time complexity of \( D \) for broadcast. We will study distributed algorithms for constructing a BFS later.
3.4 Convergecast on a Tree

Convergecast can be considered the “opposite” of broadcast, where the goal is to collect information from all nodes on a tree to the root of a spanning tree in an “upwards” fashion. Convergecast is a very useful algorithm in many applications.

Let $T$ be a spanning tree rooted at node $r$. Assume that each node knows which of its adjacent edges points to its parent (except $r$, which has no parent). Let the height of the tree be $h$ with the root at level $h$ and the leaves at level 0. Each node $v$ has a value $a_v$ (say a number) associated with it. Suppose we want to compute the sum of all the node values in the network, i.e., $\sum_v a_v$. Then we can use convergecast to accomplish this as follows. The algorithm starts at the leaf nodes (level 0). In the first round, each leaf node sends its value to its parent (level 1). A node at level $\ell > 1$ upon receiving values from all its children at $\ell - 1$, sums up all the values received in addition to its own value; it then sends this summed value to its parent (if any). The above process terminates at the root node $r$; Exercise 3.3 asks you to show that at the end of the algorithm, $r$ has value equal to $\sum_v a_v$ and the time taken is $O(\text{depth}(T))$. This is a useful application of convergecast called as aggregation. Figure 3.2 illustrates convergecast.

Figure 3.2: Illustrating the Convergecast algorithm in a graph. $s$ is the source node.

Another important application is termination detection. Consider the flooding algorithm. Suppose the source wants to know when the algorithm terminated. This can be achieved as follows. First, note that flooding can be used to construct a spanning tree rooted at the source (discussed in Section 3.7). Let $T$ be the spanning tree constructed. Each leaf node in the tree will send an “echo” message
to its parent; each internal node (except the root) when it receives echo messages from all its children will send an echo message to its parent; the root when it receives echo messages from its children will know that the broadcast algorithm has terminated successfully.

### 3.4.1 Pipelining

Consider the following modification of the sum finding problem: each node has a list of $k$ values, i.e., each node $i$ has $<a_i(1), \ldots, a_i(k)>$, and the goal is to compute the list of (component-wise) sums

$$\left(\sum_{i=1}^{n} a_i(1), \ldots, \sum_{i=1}^{n} a_i(k)\right)$$

at a source $s$. Again, we assume that there is a spanning tree rooted at $s$.

A straightforward convergecast algorithm applied to each component value in sequential order, i.e., performing $k$ successive convergecasts one after the other, takes $O(k \times \text{depth}(T))$ time (assuming that each node knows the $\text{depth}(T)$ value and can wait for that much time between convergecasts). Using a technique called pipelining, one can do much faster. The basic idea for pipelining is simple: whenever a node can send a message through an edge (i.e., there is no congestion) it sends it. In convergecast, pipelining can be applied as follows: each node when it receives the values will send the aggregated values component-by-component to their respective parent. More precisely, each leaf, in round $j$ ($1 \leq j \leq k$), sends its $j$th component value in its list to its parent; each intermediate node $v$ when it receives the $j$th component values from its children (in round $j + \text{height}(v)$, where $\text{height}(v)$ is the height of node $v$ in tree $T$; leaves are at height 0) will aggregate these values and sends it to its parent. Using an inductive argument, the total number of rounds for the pipelined convergecast can be shown to be $O(k + \text{depth}(T))$ (Exercise 3.4).

### 3.5 Upcast

We are given $k$ items distributed arbitrarily on the nodes of a network $G = (V, E)$. A node can have zero, one, or more items. (We assume that each item is of small size, i.e., can be represented in $O(\log n)$ bits.) The goal in the upcast problem is that we would like to collect all these items at a node $s \in V$. Assume that there is a spanning tree $T$ rooted at $s$. Note that, unlike in the convergecast setting, one cannot aggregate items; all the items have to be individually collected at the root. However, one can use the convergecast algorithm to solve the problem in a
trivial way: convergecast each item sequentially one by one to the root, waiting for \( \text{depth}(T) \) time between convergecasts; this takes \( O(k \times \text{depth}(T)) \) time. We next describe a simple algorithm (Algorithm 2) that uses pipelining to achieve an \( O(k + \text{depth}(T)) \) time.

Assume that each item has a unique number associated with it, say its ID (IDs as usual, will be represented using \( O(\log n) \) bits). At any round, each node does the following: among the set of items that it currently has, send the item with the highest ID to its parent (the sent item is then removed from its set). The main idea of this algorithm is to use the ID to break ties; it gives priority to the item that is ranked highest. This kind of priority is very useful in analysis to show bound on the running time. Figure 3.3 shows an illustration of the algorithm. The following theorem establishes the correctness and time complexity of this upcast algorithm.

**Theorem 3.4.** Algorithm 2 upcasts all the items to the root and takes \( O(k + \text{depth}(T)) \) time.

**Proof idea:** The highest ranked item (i.e., the item with the highest ID) — call it \( z \) — is never delayed by any other item, since it has the highest priority. Hence it reaches the root in exactly \( d_z \) steps, where \( d_z \) is its distance from the root. The second-highest ranked item — call it \( y \) — can be delayed by only the highest ranked item. Furthermore, it is important to note that it can be delayed at most once by the highest ranked item. Why? Consider the situation when \( z \) is delayed by \( y \). This means both these items are in the same node \( v \) at some round and \( z \) has to be forwarded to \( v \)'s parent before \( y \). Once \( z \) is forwarded, it
is always “one-step” ahead of $y$ and never again delays $y$. Thus $y$ reaches the root in at most $d_y + 1$ steps, where $d_y$ is its distance from the root.

**Proof.** Consider the $j$th highest item (we refer to the item and its rank interchangeably). It can possibly be delayed only by items ranked higher than itself, $j - 1$ other items. Let $j' (1 \leq j' \leq j - 1)$ be the rank of the lowest ranked item that delays $j$. Note that once this happens, it is never delayed again, by $j'$; any further delay can only be attributed to an item that is ranked higher than $j'$ (say $j''$): this is because, $j''$ can delay $j'$ by one round which will allow $j$ to “catch” $j'$ which incurs an additional delay of one round for $j$; however, this delay can be “charged” (attributed) to $j''$.

It is left as an exercise to show that the above bound is optimal, i.e., every upcast algorithm needs $\Omega(k + \text{depth}(T))$ rounds in the worst case.

Exercise 3.5 asks you to show that an even simpler rule (no priority is needed) also gives the same time bound for upcast. This rule thus applies even where there is no global ordering of the items.

### 3.6 Downcast

The objective of downcast is the opposite of upcast: Given $k$ distinct items at the root (source nodes) $s$ of a (spanning) tree $T$, the goal is to disseminate all the $k$ items to all the nodes in the tree. As in upcast, using pipelining, one can show that downcast can be accomplished in $O(k + \text{depth}(T))$ time. See Worked Exercise 3.1.

**Algorithm 2** An upcast Algorithm; code executed by node $v$. Initially there are $k$ items distributed among the nodes in the network.

1. Let $S(v)$ be the initial set of items that node $v$ possesses.
2. Let item $a$ be the item with the highest ID in $S(v)$.
3. Send $a$ to parent of $v$ in the tree $T$.
4. $S(v) = S(v) - v$.

### 3.7 Constructing a BFS spanning tree

The Flooding algorithm can be modified to construct a breadth-first spanning tree in the network. The following algorithm constructs a BFS tree rooted at
3.7. CONSTRUCTING A BFS SPANNING TREE

Before we discuss the algorithm, it will be instructive to understand what is the “output” that we expect from a distributed algorithm. At the start of the algorithm, each node only has local knowledge, i.e., it knows only about itself (its internal state) and its incident edges (i.e., the port numbers where these edges connect). At the end of the distributed algorithm, we expect each node to have a “local” view of the constructed BFS tree as well. More precisely, we would like each node to know which of its neighbours is its parent (if any) in the tree and which of its neighbours are its children (if any). Note that the root will not have a parent and leaf nodes will not have any children. Note that each node (except) the root will have exactly one parent. Note that this information is all that is needed for a node to “route” information in the tree, e.g., if each node repeatedly forwards information to its respective parent, eventually the root will receive it. On the other hand, as we saw in Tree broadcast, if each node sends its information to all its children, then broadcast can be implemented.

3.7.1 Distributed BFS tree construction algorithm

Source node $s$ sends an “invite” message inviting its neighbors to be its children. When a node $v$ receives an “invite” message from one or more of its neighbours for the first time, it will respond to exactly one such invite message from its neighbor and chooses that neighbour to be its parent node; it does so by sending an “accept” message. Any invite message that it subsequently gets (in a later round) will be ignored. A node $u$ that receives “accept” message from its neighbor(s) will designate those nodes to be its children. A node finishes its execution after processing accept messages (if it gets any). See Algorithm 3 and an example illustration in Figure 3.4.

Algorithm 3 BFS Tree Construction Algorithm; code executed by node $v$.

1: if $v = s$ then
2: send “invite” message to all neighbours
3: designate all neighbors as child nodes
4: else if when “invite” message is received for the first time then
5: send “accept” message to exactly one neighbor (say $p$) that sent an invite message and designate $p$ as parent
6: send “invite” message to all neighbors except $p$.
7: when “accept” message is received from any neighbor $u$, designate $u$ as child
Figure 3.4: BFS tree construction algorithm on a graph starting from source node S. Figures (1)-(6) show the steps of the algorithm. The BFS tree constructed is shown. The table shows the parent-child relationships and the invite-accept steps executed.

Analysis of the BFS tree algorithm

The correctness of the algorithm can be established by induction, in a way that is very similar to the Flooding algorithm. This is left as an exercise.

Let’s now analyse the complexity of the algorithm. It is clear that the message and time complexity is asymptotically equivalent to the message complexity of Flooding, i.e., \(O(|E|)\) and \(O(D)\) respectively.

### 3.7.2 Termination Detection

An important issue in distributed algorithms is termination detection, i.e., one or more nodes in the network knowing when the algorithm has terminated.
3.7. CONSTRUCTING A BFS SPANNING TREE

Figure 3.5: Illustrating termination detection. Steps (1)-(4) show the echo sending (convergecast) process. Once the echos reach the root, the source node broadcasts “done” message to all the nodes (via flooding). The broadcasting is shown in steps (5)-(7).

all the network. Note that termination detection is a “global” knowledge, i.e., requires knowledge about the status of all nodes in the network — in particular, the knowledge of the time when all nodes have finished executing the algorithm. In the BFS example, s, the root node might need to know when the BFS tree has been fully constructed. This might be useful in various applications, e.g., if s wants to use the tree to do a subsequent Tree broadcast. Although constructing a BFS tree has asymptotically the same complexity as Flooding, once a tree has been constructed, it can serve as a “backbone” for repeatedly broadcasting messages.

How can termination detection be done? The idea is to do Convergecast, as mentioned earlier. Each leaf node will send an “echo” message to its parent.
Note that a node will know that it is a leaf, when it sees that it did not receive any “accept” messages from any of its neighbors. Each internal node, will wait till it receives echo messages from all its children before it sends an “echo” message back to its parent. When s receives echo messages from all its children, it knows that the algorithm has terminated. It can then proceed to inform all other nodes of the termination by doing a Tree broadcast — e.g., by sending a “done” message. Figure 3.5 illustrates termination detection on an example graph.

3.8 Information Spreading

The information spreading/dissemination problem (also sometimes called as the gossip problem) can be considered as a generalization of the upcast problem and can be stated as follows. Given a network \( G = (V, E) \) of \( n \) nodes and \( m \) edges, and a set of \( k \) tokens (which can be considered as small-sized items of size \( O(\log n) \) bits each) arbitrarily distributed among the \( n \) nodes, the goal is to disseminate the \( k \) tokens to all the \( n \) nodes. The information spreading problem is a generalization of upcast because instead of sending all the \( k \) tokens to one root node, here we need to send it to all nodes. Furthermore, a spanning tree is not assumed to be given apriori. The information spreading problem has many applications. For example, it can be used to acquire global knowledge of the network. More precisely, suppose every node wants to know the global topology. Then each node can send the information about its incident edges (the two endpoints of each incident edge) to all nodes in the network. In this scenario, we have \( d(v) \) tokens for each node (each token corresponding to an edge) and hence the total number of tokens is \( k = \sum_{v \in V} d(v) = 2m \).

An algorithm for information spreading We focus on optimal algorithms for information spreading algorithm. We leave it as an exercise to show that \( \Omega(D + k) \) is a lower bound on the running time of any information spreading algorithm (this is similar to the upcast lower bound). Hence, as far as time optimality is concerned, the goal is to obtain an \( O(n + k) \) round algorithm. One way of achieving this is as follows. Find a BFS tree (using Algorithm [3]). Then upcast all the \( k \) items to the root of the tree and the finally downcast all the items to all the nodes. The time complexity of BFS is \( O(D) \); upcast and downcast takes \( O(D + k) \) time each. Hence overall time is \( O(D + k) \).

Regarding message complexity, it is clear that \( \Omega(nk) \) is a lower bound, since each token has to reach all the \( n \) nodes. Let’s calculate the message complexity of the above method that uses BFS, upcast, and downcast. BFS takes \( O(m) \)
messages, upcast takes $O(kD)$ messages, and downcast takes $O(nk)$ messages; hence overall the message complexity is $O(m + nk)$. Since $\Omega(m)$ is a lower bound for broadcasting one token to the entire network (we will show this bound later), this is an optimal bound.

3.9 Worked Exercises

**Worked Exercise 3.1.** Show that downcast of $k$ items in a tree $T$ can be accomplished in $O(k + \text{depth}(T))$ rounds.

**Solution.** Each item has a unique ID, and there exists a total order on these IDs. The items are sent down in decreasing order of their ID. Any item takes at least $\text{depth}(T)$ time to be sent from root to leaf. Due to congestion, an item with the $i^{th}$ highest ID must wait for $i - 1$ rounds before it can be sent from the root. At subsequent nodes, due to the pipelining, it is not further delayed by higher ID items. Thus each item takes at most $i - 1 + \text{depth}(T) \leq k - 1 + \text{depth}(T) = O(k + \text{depth}(T))$ rounds to reach all leaves from the root. ■

**Worked Exercise 3.2.** Consider the following algorithm for information spreading. Assume that each token has a priority number. Each node simply floods each token separately; if there are multiple tokens to send, then one with the highest priority is sent. More precisely, when a node receives a token (from any of its neighbors) for the first time it saves it in its buffer. In every round, it picks a token with the highest priority among those in its buffer (if its non-empty) and sends it to all its neighbors and then deletes it from its buffer. Show that this algorithm will take $O(D + k)$ rounds.

**Solution.** Consider any token $a$. Let it be initially at node $u$. Consider any node $v$. We argue that in at most $D + k$ rounds, $a$ will reach $v$. Consider the shortest path between $u$ and $v$ in the graph. If token $a$ has the highest priority then it is not delayed at all. As argued for the priority-based upcast algorithm (Algorithm 2), a token delay can be attributed to a higher priority item at most once. Hence the time bound. ■

3.10 Exercises

**Exercise 3.1.** Given a network $G$, for any source node $s$ in $G$, there exists a node which is at a distance of at least $\Omega(D)$ from $s$, where $D$ is the diameter of the network.
Exercise 3.2. Prove Theorem 3.3.

Exercise 3.3. Show that the convergecast algorithm described in Section 3.4 correctly calculates the $\sum_v a_v$ in time $O(\text{depth}(T))$.

Exercise 3.4. Show that the pipelined convergecast algorithm described in Section 3.4.1 correctly calculates component-wise sums in time $O(k + \text{depth}(T))$.

Exercise 3.5. Consider an upcast algorithm that where each node (except the root) maintains a list of items to be sent to its parent. There are a total of $k$ items over all the nodes. In every round, a node sends an arbitrary item in its current list to its parent and deletes it from the list. Show that this upcast algorithm terminates in $O(k + \text{depth}(T))$ time.

Exercise 3.6. Prove the correctness of the distributed BFS tree construction algorithm.

Exercise 3.7. Let $h$ be the depth (or height) of the BFS tree constructed by the algorithm. Show that the time complexity of the BFS algorithm is $O(h)$. Show that $h$ is related to the diameter $D$ of the network as follows: $D / 2 \leq h \leq D$.

Exercise 3.8. Incorporate termination detection in Algorithm 3.

Exercise 3.9. Show that upcast of $k$ items in a tree $T$ needs at least $\Omega(k + \text{depth}(T))$ rounds.

Exercise 3.10. Consider the following algorithm for information spreading which is even simpler than the one described in Worked Exercise 3.2. Each node simply floods each token separately. More precisely, when a node receives a token (from any of its neighbors) for the first time it saves the token in its buffer. In every round, a node picks a token from its buffer (if its non-empty) and sends it to all its neighbors and then deletes the token from its buffer (and hence each token is sent only once). Show that this algorithm will take $O(D + k)$ rounds.

Exercise 3.11. In this problem, we will design a distributed algorithm for constructing a depth-first search (DFS) spanning tree (see Appendix D) in a given network $G$ (as usual, we will assume that $G$ is connected and undirected) with $n$ nodes and $m$ edges. We will assume the synchronous CONGEST model, as usual.

The sequential DFS algorithm is as follows. Given a starting node $s$, whenever the search reaches a node $v$ (initially, $v$ is $s$ itself), the next node to visit is decided as follows. If $v$ has a neighbor, say $u$, that has not been visited yet, then $u$ is visited next. Otherwise, the search returns to the node from which $v$ was visited for the first time. If there is no such vertex, i.e., $v$ is $s$ itself, then the search finishes. The
DFS spanning tree is the tree consisting only of those edges in the graph which are used to visit vertices for the first time in the DFS algorithm.

(a) Describe a natural distributed implementation of the above sequential algorithm. What is the time and message complexity? (Hint: Note that at any round, there is only one “active” node in the algorithm, the one that is currently being visited.)

(b) Show how to get a distributed implementation with time complexity $O(n)$. 
Chapter 4

Distributed Shortest Paths Algorithms

Routing is a fundamental operation in a distributed network. Consider the
Internet, a distributed wide area communication network interconnecting various
end systems or hosts by a network of communication links and packet switches
(e.g., routers). A packet switch forwards a packet arriving on one of its incoming
communication links to one of its outgoing communication links. The sequence
of communication links and packet switches traversed by the packet, starting
from the source to the destination, is known as a route or a path through the
network. The Internet employs different routing algorithms or protocols to
route packets (data). These routing protocols help in configuring the routing
(or forwarding) tables of the routers which indicates to which of the neighbors
a packet is to be forwarded based on its destination address. Each node has a
routing table that shows for each destination node in the graph (which typically
consists of all other nodes in the graph) the next hop node to forward the
message (packet). Figure 4.1 shows the routing tables for an example graph
and gives an illustration.

A routing algorithm’s goal is to find a “good” path or route to send data from
a source node to a destination node. Typically, a good path is one that has the
least cost. Consider a weighted network \( G = (V, E, c) \) with positive real-valued
edge costs given by the cost function \( c \). The cost of a path \( p = (e_1, \ldots, e_k) \) is
defined as \( c(p) = \sum_{i=1}^{k} c(e_i) \). As mentioned in Chapter 2, costs can model delay,
congestion, length etc.

For a source-destination pair \( (s, t) \in (V \times V) \), the goal is to find a least-cost
(or shortest) path, i.e., a path from \( s \) to \( t \) in \( G \) that has minimum cost. If all edges
in the graph have cost 1, the shortest path is the path with the smallest number
CHAPTER 4. DISTRIBUTED SHORTEST PATHS ALGORITHMS

of edges between the source and destination. For example, the Internet uses least-cost path routing. Finding shortest paths is one of the most fundamental and important distributed network problems and many algorithms have been proposed. In this chapter, we focus on two basic algorithms, motivated, in part, by real-world networks, especially the Internet.

There are two types of shortest paths algorithms according to whether they are global or local:

- A global shortest path algorithm computes the least-cost path between a source and a destination using complete, global knowledge of the network. In networking terminology, it happens to be referred to as a link state (LS) algorithm.

- A distributed shortest path algorithm computes the least-cost path in an iterative, distributed fashion. No node has complete information about all the edge costs. In networking terminology, it happens to be referred to as a distance vector (DV) algorithm.

The link state and distance vector algorithms are essentially the routing algorithms used in the Internet today. There are two types of protocols based on whether the routing is made within an Autonomous system (intra-AS routing) or between Autonomous systems (inter-AS routing). Shortest path routing is typically used for intra-AS routing. A protocol called Border Gateway Protocol (BGP) is used for inter-AS routing.

4.1 A Global Algorithm

A global routing algorithm assumes knowledge of the global network topology and costs of all the edges. This can be accomplished by information spreading (cf. Section 3.8). One way to gather this global information is by having each node broadcasts its identity number and costs of its incident edges to all other nodes in the network using a broadcasting algorithm, e.g., flooding (cf. Exercise 3.10). (An edge information consists of its cost and the identities of its two end nodes.) At the end of this broadcasting, each node will know the identities of all other nodes in the network as well as the incident edges (and their costs) of all nodes — this means that each node has complete knowledge of the entire graph. Thus, each node can run a (centralized) shortest path algorithm and compute the same set of shortest paths as any other node. A well-known shortest path algorithm is the Dijkstra’s algorithm (e.g., see [3]). Figure 4.2 shows an illustration of this Dijkstra’s algorithm on a graph. This algorithm takes as input a weighted
4.2. THE DISTRIBUTED BELLMAN-FORD ALGORITHM

graph \( G = (V, E, c) \), a source vertex \( s \in V \) and computes shortest paths (and their values) from \( s \) to all nodes in \( V \). (In principle, any graph problem can be solved by this approach, since all nodes have access to the global topology information.) This computation is done locally (within each node) and incurs no communication.

The message complexity and time complexity of the global routing algorithm is solely determined by the broadcast algorithm. We note that each edge information constitutes a message (the cost and the two endpoints of an edge can be represented using \( O(\log n) \) bits). Thus the goal is to broadcast all the edges (and their respective costs) to all nodes in the networks. Broadcast can be done by flooding (Section 3.1.1); recall that, in flooding, a node never sends the same message twice. If broadcast is done by flooding (cf. Section 3.1.1), then every node's local information (i.e., the cost and identities of the endpoints all its incident edges) then the message complexity is \( O(|E|^2) \), since \( O(|E|) \) messages have to be broadcast, each of which causes \( O(|E|) \) messages to be sent by flooding. The time complexity is clearly upper bounded by \( O(|E|D) \) (where \( D \) is the diameter of the network), since flooding one message takes \( O(D) \) time (cf. Chapter 3) and there at most \( O(|E|) \) messages. This is a pessimistic upper bound. Exercise 4.1 asks you to show a better time bound.

Internet’s Open Shortest Path First (OSPF) protocol uses a LS routing algorithm as mentioned above. Since the LS algorithm is centralized, it may not scale well when the networks become larger. This is also an issue when nodes have limited resources e.g., computation power or memory, which impacts the power of local computation. In this book, we will generally ignore the complexity of “within-node” computation, but this is an issue in practice; in any case, it will be desirable to design algorithms that has low “within-node” computation cost.

Exercise 4.4 discusses a distributed implementation of Dijkstra’s algorithm.

4.2 The Distributed Bellman-Ford Algorithm

The algorithm that is described below is called the distributed Bellman-Ford (DBF) algorithm. It is one of the most well-known and classical algorithms that is also naturally viewed as a distributed algorithm. It is used in the Routing Information Protocol (RIP) and the Border Gateway Protocol (BGP) of the Internet.

We will describe the basic idea behind the DBF algorithm (cf. Algorithm 4). Suppose we want to compute the shortest path between \( s \) and all other nodes in a given undirected graph \( G = (V, E, c) \) with real-valued positive edge weights.

\footnote{Nodes need to communicate with their neighbors first to obtain the information of the identities of nodes at the other end of their incident edges; this takes 1 round and \( O(|E|) \) messages.}
This is called as the *single-source shortest paths* problem. (This can be extended to compute shortest paths between every pair of nodes, as discussed later.)

During the algorithm each node \( x \) maintains a distance label \( a(x) \) which is the current known shortest distance from \( s \) to \( x \), and a variable \( p(x) \) which contains the identity of the previous node on the current known shortest path from \( s \) to \( x \). Initially, \( a(s) = 0 \), \( a(x) = \infty \), and \( p(x) \) is undefined for all \( x \neq s \). When the algorithm terminates, \( a(x) = d(s, x) \), where \( d(s, x) \) is the shortest path distance between \( s \) and \( x \), and \( p(x) \) holds the neighbor of \( x \) on the shortest path from \( x \) to \( s \). Thus, if node \( x \) wants to route to \( s \) along the shortest path it has to forward its data to \( p(x) \).

The DBF consists of two basic rules: the update rule and the send rule. The update rule determines how to update the current label according to a message from a neighboring node. The send rule determines what values to send to its neighbors and is applied whenever a node adopts a new label.

**Update rule:** Suppose \( x \) with a label \( a(x) \) receives \( a(z) \) from a neighbor \( z \). If \( a(z) + c(z, x) < a(x) \) (where \( c(z, x) \) is the cost of edge \((z, x)\)) , then it updates \( a(x) \) to \( a(z) + c(z, x) \) and sets \( p(x) \) to be \( z \). Otherwise \( a(x) \) and \( p(x) \) are not changed.

**Send rule:** Let \( a(x) \) be a new label adopted by \( x \). Then \( x \) sends \( a(x) \) to all its neighbors.

**Algorithm 4** Distributed Bellman-Ford Algorithm; code executed by node \( x \) to compute shortest paths from a source node \( s \) in Network \( G = (V, E, c) \).

```plaintext
1: if \( x = s \) then
2: \( a(x) = 0; p(x) = s \)
3: else
4: \( a(x) = \infty; p(x) = \text{undefined} \)
5: Send \( a(x) \) to all neighbors
6: On receiving a message containing \( a(z) \) from a neighbor \( z \):
7: if \( a(z) + c(z, x) < a(x) \) then
8: \( a(x) = a(z) + c(z, x); p(x) = z \)
9: Send \( a(x) \) to all neighbors
```

### 4.2.1 Correctness and Analysis of the DBF algorithm

As usual, we assume the synchronous CONGEST model. The following theorem gives the correctness and complexity of the DBF algorithm.

**Theorem 4.1.** The DBF algorithm terminates after at most \( n \) rounds. When it terminates, \( a(x) = d(s, x) \) for all nodes \( x \). The message complexity is \( O(n|E|) \).
4.2. The Distributed Bellman-Ford Algorithm

Proof. Fix a node \( x \in V \), we prove that the algorithm computes a shortest path from \( s \) to \( x \).

Let \( P = v_0, v_1, \ldots, v_k \), where \( v_0 = s \) and \( v_k = x \) be a shortest path from \( s \) to \( x \). Note that \( k < n \); this is because, a shortest path cannot have cycles, and hence has to be simple, i.e., it can at most \( n \) nodes on the path.

We prove by induction on \( i \) that after the \( i \)th round, the algorithm has computed the shortest path from \( s \) to \( v_i \), i.e., \( a(v_i) = d(s, v_i) \).

The hypothesis holds for \( v_0 = s \) in round zero.

Assume that it holds for \( j \leq i - 1 \). After the \( i \)th iteration,

\[
a(v_i) \leq a(v_{i-1}) + c(v_{i-1}, v_i)
\]

which is the shortest path from \( s \) to \( v_i \), since \( P \) is a shortest path from \( s \) to \( v_i \), and the right hand side is the distance between \( s \) to \( v_i \) on that path. Hence after \( i \) rounds, \( a(v_i) \) has the correct shortest path distance from the source \( s \).

Thus after \( n \) rounds, shortest paths from \( s \) to every other node has been computed.

Since, in each round \( O(|E|) \) messages are exchanges, the total message complexity is \( O(n|E|) \).

From the above proof, it can be seen that the algorithm will compute the correct values even if the nodes operate asynchronously, i.e., delays don’t matter (we will look at asynchronous algorithms in more detail later). An important observation is that the algorithm is “self-terminating” — there is no signal that the computation should stop; each node just stops when the correct shortest path values are reached (see Exercise 4.1).

4.2.2 Computing All-pairs Shortest Paths

The DBF algorithm can be generalized to compute the shortest path between all pairs of nodes, by maintaining in each node \( x \) a distance label \( a_y(x) \) for every \( y \in V \). This is called the distance vector of \( x \) (especially in networking parlance). Each node stores its own distance vector and the distance vectors of each of its neighbors. Whenever the distance vector changes, the node will send its distance vector to all of its neighbors. The receiving nodes then update their own distance vectors according to the update rule.

4.2.3 “Count-to-Infinity” Issue

A drawback of the DBF algorithm is that convergence time can be made arbitrarily large if the initial distance labels are not correct. A typical scenario when this
can happen is when there is some change in the network (after the algorithm has started or converged); this is a common feature in real-world dynamic networks, where the graph topology or weights can change over time.

Consider the following simple network consisting of 4 nodes and 3 links shown in Figure 4.4. The goal is to compute the shortest paths to D. Initially each link has weight 1 and each node had calculated the shortest path distance to D. Thus the distances of A, B, and C to D are 3, 2, and 1 respectively. Now suppose the weight of edge (C, D) changes from 1 to a large positive number, say L. Assuming a synchronous behavior, in the subsequent iteration, C will set its distance label to D as 3, since B supposedly has a path to D of length 2. In the following iteration B will change its distance estimate to 4, since the best path it has is through C. This will continue until C distance label reaches L. Thus the number of iterations taken to converge to the correct shortest path is proportional to L, which can be much larger than n, the number of nodes in the network. This problem, is referred to as the “count-to-infinity” issue and shows that the algorithm can be very slow in reacting to a change in an edge cost.

4.3 Worked Exercises

**Worked Exercise 4.1.** Consider a connected graph with positive edge weights. In the Distributed Bellman-Ford algorithm, suppose a node does not update its a(·) value (i.e., its current distance estimate) in a round. (We ignore the rounds when the node’s a(·) value is ∞.) Does it mean the algorithm has terminated as far as this node is concerned. Does your answer depend on whether the graph is weighted or unweighted?

**Solution.** The answer depends on if the graph is weighted or unweighted. We first show the statement is true for unweighted graphs and then show it is false for weighted graphs.

For an unweighted graph, for a given non-source node v, its a(·) value is updated only once when a path from the root to it of length < ∞ is found. This is because distance here is equivalent to the number of nodes in the path from source to v. The first path to v that is found will be of shortest distance from the root. Subsequent paths will only be longer, and thus the distance to the root along those paths is correspondingly longer. Thus v’s a(·) value is updated only once over the course of the algorithm and the claim is true.

For a weighted graph, we construct a counterexample. Consider a graph with two paths from the source to a given node v. Let the number of nodes in the first path be 2 and the number of nodes in the second path be 7. Further, let
4.4. EXERCISES

37

the total weight of edges in the first path be 100 and the second path be 8. Now, 
v will update its \( a(\cdot) \) value in the 2nd round and update its \( a(\cdot) \) value again only 
in the 7th round, thus showing the claim to be false.

4.4 Exercises

Exercise 4.1. Show that broadcast of the entire graph topology by flooding as 
 discussed above takes \( O(|E| + D) \) time. Hint: See Exercise 3.10.

Exercise 4.2. The shortest path diameter between two nodes \( u \) and \( v \), denoted by 
\( S(u, v) \), is defined as the number of edges in a shortest path between \( u \) and \( v \); if 
there is more than one shortest path, the one with the minimum number of edges 
is taken to be \( S(u, v) \). The shortest path diameter of the network, denoted by \( S \), is 
defined as \( S = \max_{u,v} \{S(u, v)\} \).

- Show that in an unweighted graph, \( S \) is the same as the diameter \( D \) of the 
  network.

- Given an example of a weighted graph where \( S \) is different from \( D \). Is \( S \) 
always larger than \( D \), the network's diameter?

- Show that the Distributed Bellman-Ford algorithm actually terminates in at 
  most \( S \) rounds in any graph \( G \).

Exercise 4.3. Can you suggest some method to handle the count-to-infinity issue? 
Discuss the pros and cons of your method.

Exercise 4.4. In this problem, we look at a distributed implementation of 
Dijkstra's algorithm (see e.g., [3]) for finding the single source shortest paths, 
i.e., finding shortest paths (and distances) from every node to a given source node 
\( s \). We assume that we are given a connected weighted (undirected) network with 
positive weights on the edges. We will assume the synchronous CONGEST model.

The algorithm is outlined as follows. The algorithm builds a shortest path tree 
\( (SPT) \) \( T \) rooted at the source \( s \). The \( SPT \) is a spanning tree of the network with the 
property that the path from \( s \) to any node \( v \) on the tree is the shortest path from \( s \) 
to \( v \) (see Figure 4.2 for an illustration).

The source \( s \) (which will be the root of the \( SPT \)) initiates a "join" message. The 
algorithm proceeds in phases. In each phase, a new node is added to the tree. In 
the beginning (phase 0), there is only one node in the tree \( T \) which is \( s \) itself. In 
phase \( i \geq 1 \), the \( i \)th nearest node to \( s \) will be added to the tree. The algorithm will 
finish after \( n - 1 \) phases, when all nodes are added to the tree. Let \( T_p \) denote the 
tree at the beginning of phase \( p \). Each node \( v \) in the tree maintains a value \( d(v) \)
which is the (shortest) distance from \( v \) to \( s \) and its parent on the tree. All nodes not in the (current) tree also maintain a \( d(v) \) value which is the (shortest) distance to \( s \) via a path that uses only tree nodes of \( T_p \); if no such path exists then \( d(v) = \infty \). Each non-tree node also maintains a parent node which is the tree node neighbor in \( T_p \) via which the shortest path to \( s \) exists (using only tree nodes); if there is no such path, then it is undefined. Upon termination, note that the parent gives the next-hop neighbor on the shortest path to the source.

A phase consists of the following:

1. The root broadcasts a “new phase” message on \( T_p \).
2. When a node in the tree receives a “new phase” message it sends it to all its neighbors.
3. A non-tree node when it receives a “new phase” message replies “non-tree node” and also sends its \( d(v) \) value to the tree node(s) that sent the “new phase” message.
4. All tree nodes that receive replies from a non-tree neighbor convergecast the minimum value of \( d(v) \) and the ID of the non-tree node that sent it.
5. Once the convergecast terminates, the root has the minimum \( d(v) \) value and the ID of the non-tree node that has this minimum value. The root broadcasts this value and ID to all nodes in the tree.
6. Once the non-tree node hears its own ID, it adds itself to the tree (becomes a tree node). It then sends “update” message to its neighbors.
7. A non-tree node \( u \) that receives an update message from its tree neighbor \( v \) updates its \( d(u) \) value if \( d(v) + w(u,v) < d(u) \); in that case, \( u \) sets its \( d(u) \) value to be \( d(v) + w(u,v) \). It also updates its parent to \( v \). It then sends a “done” message to its tree neighbor(s). The “done” messages are convergecast to the root.
8. Once the root receives a "done" message, it initiates a new phase.

- Argue that the above algorithm correctly implements Dijkstra’s algorithm.
- Analyze the time and message complexity of this algorithm.
Figure 4.1: The figure shows routing tables in each node in a graph. (If slot is empty in a routing table, then it means it is a neighbor.) For example, if node B wants to route a message to node E, then according to B’s routing table the next hop is node D and so it will forward the message to D (along with the information of the destination node, which is E). According to D’s routing table, E is a neighbor, so it will forward to E directly.
Figure 4.2: Illustrating Dijkstra’s algorithm (centralized) on a graph. The algorithm computes shortest paths from A to all nodes in the graph; the output can be represented as a shortest path tree (SPT). The final SPT is shown — the (unique) path in the tree from A to any other node gives the shortest path to that node.
4.4. EXERCISES

Figure 4.3: Illustrating Belman-Ford distributed algorithm on a graph. The algorithm computes shortest paths from A to all nodes in the graph; the output can be represented as a shortest path tree (SPT). The final SPT is shown — the (unique) path in the tree from A to any other node gives the shortest path to that node.
Figure 4.4: An example network for count-to-infinity problem.
Chapter 5

Leader Election

Leader election is one of the fundamental tasks in distributed computing. It is one of the fundamental “symmetry breaking” problems that we will study. As mentioned in Chapter 2 (also see Section 3.1.1), all nodes in the network will execute the same distributed algorithm, starting with the initial state. The only node-specific information that can influence the behaviour of each node is its (unique) ID and the random choices it makes (this will be case in randomized algorithms, as explained below). In symmetry breaking, the goal is to break the initial starting symmetry among all nodes (due to the execution of the same algorithm, starting from the same initial state). Hence at the end of symmetry breaking, different nodes will end up exhibiting more than one type of behavior or decision. Leader election is one such problem, where the goal is to elect an unique leader of the entire network. That is at the end of leader election, exactly one process knows that it is the leader; all others know they are not.

Leader election is a basic primitive that serves in many applications. In many applications, a leader node is required that performs tasks on behalf of other nodes or the entire network. In one of the first applications, leader election was used to decide who has the right to own a token in a “token-ring” network (where the network is a cycle). Only one node can have the token at any point in time, and that node has the right to transmit a message, while other nodes should not. More generally, many nodes might contend to access some shared resource and to break this symmetry, they elect a leader which only has to right to some shared resource at any point in time.

As usual, we consider a network of $n$ nodes, represented as an undirected connected (not necessarily complete) graph $G = (V,E)$. We will assume the synchronous CONGEST model. We assume that all nodes have unique identifiers; the identifiers can be arbitrary numbers which can be represented by $O(\log n)$
CHAPTER 5. LEADER ELECTION

bits. We will assume that all nodes are awake initially and start executing the algorithm simultaneously — so called simultaneous wakeup model. However, our algorithms and analysis can also be adapted to the adversarial wakeup model where the nodes are awoken at arbitrary points in time, with the restriction that nodes wake upon receiving a message and at least one node is initially awake.

We now define the leader election problem formally. Every node \( u \) has a special variable \( \text{status}_u \) that can be set to a value in \( \{\perp, \text{NON-ELECTED}, \text{ELECTED}\} \); initially \( \text{status}_u = \perp \). An algorithm \( A \) solves leader election in \( T \) rounds if, from round \( T \) on, exactly one node has its status set to \( \text{ELECTED} \) while all other nodes are in state \( \text{NON-ELECTED} \). These nodes need not be aware of the identity of the leader. In this book, we will focus on this implicit variant of leader election. In another variant, all the non-leaders may change their \( \text{status} \) component to the value \( \text{non-leader} \), and moreover, every node must also know the identity of the unique leader. This formulation may be necessary in problems where nodes coordinate and communicate through a leader. This explicit leader election can be achieved by simply executing an (implicit) leader election algorithm and then broadcasting the leader’s identity using an additional \( O(n) \) messages and \( O(D) \) time (where \( D \) is the diameter of the graph).

The goal is to design efficient leader election algorithms — algorithms that have small message and time complexity. We will first discuss algorithms for two special type of networks: ring and the complete graph. Later we will focus on algorithms that work on arbitrary (general) graphs. We will first talk about deterministic algorithms (i.e., algorithms that don’t make random choices) and then focus on randomized algorithms. This chapter will thus introduce randomization in distributed algorithms which will be a frequently occurring theme for the rest of the course.

Before we discuss algorithms, we make note of a few important assumptions. First, it is essential that nodes have unique identifiers — this is essential for deterministic leader election. For deterministic algorithms, other than unique identifiers, there is no other resource to break symmetry, since all nodes execute the same algorithm starting with the same starting state. Thus in anonymous networks, i.e., in networks where nodes don’t have IDs, deterministic leader election is not possible. (On the other hand, for randomized algorithms, nodes can presumably break symmetry via random choices, as we will see below.) Second, we assume that the identifiers, while being unique, are drawn from a large enough set. In other words, our leader election algorithms should work for all IDs drawn from this set. This is needed to avoid trivial algorithms. For example, if the set of IDs is always drawn from the set \( \{1, \ldots, n\} \) and there are \( n \) nodes, then a trivial algorithm can simply assign the node with ID 1 as the
5.1 LEADER ELECTION IN A RING NETWORK

We will focus on leader election in a ring network, i.e., the network is a cycle. The cycle has been an important network historically and is also a very useful special case to design and analyze distributed algorithms. Ring network originally was used to model a token ring network, a type of local area network (LAN). In this network, nodes are arranged in the form of a cycle, and a node has the right to transmit only if it holds a unique token. If more than one node wants to transmit, they have to contend for the right to own this token; this boils down to electing a leader among the contending nodes. This is one of the earliest applications of leader election in a distributed network. As a special case, the cycle is a symmetric network (i.e., it looks the “same everywhere”), and is a good choice for studying symmetry breaking algorithms.

5.1.1 A basic algorithm

We first present a basic leader election algorithm (called BasicLE-Ring) for the ring. Although simple, the idea of this algorithm is used in subsequent leader election algorithms. This algorithm takes \(O(n^2)\) messages and \(O(n)\) rounds, where \(n\) is the size of the ring. The pseudocode is presented in Algorithm 5 (as mentioned earlier, the pseudocode is written from the perspective of a single node \(v\); all nodes execute instances of the same code). For sake of simplicity, we assume that the ring is unidirectional, i.e., every node has only one outgoing edge to its (say) clockwise neighbour. The algorithm and its analysis can be extended to apply even without this assumption. The idea of the algorithm is simple: each node sends its ID around the ring; a node (say \(v\)) forwards a received ID to its neighbor only if the received ID is smaller in value compared to its (i.e., \(v\)’s) ID. If a node receives its own ID, then it elects itself as leader. Figure 5.1 illustrates the algorithm.

Analysis

**Theorem 5.1.** Algorithm 5 is correct and has time complexity \(O(n)\) and message complexity \(O(n^2)\).

**Proof.** **Correctness.** First we analyze the correctness of the algorithm, i.e., we show that the algorithm elects a unique leader. It is clear that the node with the lowest ID will be elected as a leader. We will argue as follows. Let’s call the
Algorithm 5 BasicLE-Ring: A Basic Leader Election Algorithm for a Ring; Code for node \( v \)

1: Round 1: Send \( ID(v) \) to (clockwise) neighbor.
2: Round \( i \geq 2 \):
3: Let \( ID(u) \) be the ID received by \( v \) at the beginning of this round.
4: if \( ID(u) < ID(v) \) then
5: Send \( ID(u) \) to neighbor; \( \text{status}_v = \text{NON-ELECTED} \)
6: else if \( ID(u) = ID(v) \) then
7: \( \text{status}_v = \text{ELECTED} \)

message sent by a node as a token. The token sent by the node with the lowest ID will not be stopped by any other node. On the other hand, any other node \( v \) will not be elected as leader; its ID will be stopped by some node whose ID is smaller than \( v \). Thus one and only one node will be elected as leader.

**Complexity.** The time complexity is clearly \( O(n) \) rounds. This is because, in each round, at most one token is sent across an edge (hence no congestion in any edge); furthermore, the token from the lowest ID node goes round the entire ring once and comes back to it — this requires \( n \) rounds.

The message complexity depends on the distribution of IDs along the ring. Suppose the IDs are arranged in an increasing order in the clockwise direction, then the message complexity can be calculated as follows: the token from the lowest ID goes around the entire ring and incurs \( n \) messages, the token from the second lowest ID will be stopped only by the node with the lowest ID and hence incurs \( n - 1 \) messages (since tokens go around in a clockwise direction) and so on. Thus the total number of messages is:

\[
 n + n - 1 + n - 2 + \cdots + 1 = n(n + 1)/2 = \Theta(n^2).
\]

We note that the message complexity of the above algorithm can be better. A good case is when the IDs are arranged in decreasing order in the clockwise direction. In this case, except for the token from the lowest ID node which travels through the whole ring, the rest of the tokens travel only one hop and then get stopped. Thus the total message complexity is \( O(n) \).

### 5.1.2 An Improved Algorithm with \( O(n \log n) \) Message Complexity

Next we present another LE algorithm in a ring, a modification of the BasicLE-Ring algorithm, called ImprovedLE-Ring, that has the same asymptotic time
complexity as the basic algorithm, but has significantly improved message complexity. For this algorithm, we will assume that the ring is bidirectional or, in other words, it is undirected.

The algorithm consists of many phases, starting from phase 0 and ending in phase \( \log n \). (Unless otherwise stated, we assume that all logarithms are to the base 2.) At the beginning of phase 0, every node is in “active” status. As usual, we will describe the algorithm from the point of view of an active node \( v \) in a phase. In phase \( i \), each node \( v \) that is “active” at the beginning of this phase sends two tokens (one in each direction) up to a distance of \( 2^i \), i.e., it travels up to a node that is \( 2^i \) hops away. The origin node \( v \) also sends a “distance to travel (DTL)” parameter as part of the token, which is decremented by one when passing through each intermediate node; when DTL reaches 0, it means that the token has to return back to \( v \). Each token contains the ID of \( v \) as well. If any of the (two) tokens sent by \( v \) encounters a node along the way whose ID is smaller, then the corresponding token is stopped; otherwise it returns to \( v \) (it is possible that only one token returns and the other is stopped). Phase \( i \) lasts for \( 2 \times 2^i \) rounds, since that is the maximum time that a token takes to return to \( v \) (\( 2^i \) for the forward journey and back). If both tokens return to \( v \), then \( v \) remains active and proceeds to the next phase; otherwise, \( b \) switches its status to NON-ELECTED (but it continues to participate in the algorithm as a forwarder for messages initiated by other active nodes).

**Correctness.** First we show that the algorithm terminates with the election of a unique leader; all other nodes will set their status to NON-ELECTED. Clearly, the node with the minimum ID is elected as leader, since its token is never stopped in any of the phases. For any other node, its token will be stopped when it encounters a node whose ID is smaller than itself, in particular the smallest ID node. This is bound to happen in some phase, since the last phase goes up to a distance of \( 2^{\log n} = n \), i.e., the size of the ring.

**Complexity Analysis.** We first upper bound the number of active nodes in phase \( i \). Each such active node contributes at most \( 4 \times 2^i \) messages to the message complexity, since that is the (maximum) total trip length travelled by the two tokens initiated by this node. We make the following claim.

**Lemma 5.1.** Fix an active node \( v \) in phase \( i \). Then there is no other active node within a distance of \( 2^{i-1} \) from \( v \).

**Proof.** In phase \( i-1 \), both of \( v \)’s tokens were not killed during their trips of length \( 2^{i-1} \) from \( v \). That means that \( v \) has the smallest ID among all nodes within distance \( 2^{i-1} \). Thus any node within this distance, if it was at all active during phase \( i-1 \), its token would have been stopped by \( v \). Hence such a node cannot be active at the beginning of phase \( i \). \( \Box \)
Lemma 5.2. The number of active nodes in phase $i$ is at most $\left\lceil \frac{n}{2^{i-1}} \right\rceil$.

Proof. From Lemma 5.1, for each active node in phase $i$, there is no other active node within distance $2^{i-1}$ of it. Or in other words there will be at most 1 active node per segment of length $2^{i-1}$ in the ring. The lemma follows.

From the above two lemmas, we can analyze the message and time complexity.

Theorem 5.2. The message complexity of the ImprovedLE-Ring algorithm is $O(n \log n)$ and the time complexity is $O(n)$.

Proof. The total number of messages sent by a node is upper bounded by

$$\sum_{i=0}^{\log n} 4 \times 2^i \left\lceil \frac{n}{2^{i-1}} \right\rceil \leq \sum_{i=0}^{\log n} 4 \times 2^i \frac{n}{2^{i-2}} \leq \sum_{i=0}^{\log n} 16 \times 2^i \frac{n}{2^i} = \sum_{i=0}^{\log n} 16n = O(n \log n).$$

We leave the analysis of the time complexity as an exercise.

5.2 Randomized Distributed Algorithms

Randomization plays a central role in algorithms, probably more so in distributed algorithms. One of the fundamental uses of randomization in distributed computing is symmetry breaking. Consider the following simple setting, where there are two users $A$ and $B$ trying to use a common channel to communicate; say the users are at either end of the channel. Each user has no information on the other (e.g., it does not know the ID of the other). The constraint is that only one user can send a message in any one round. If both the users send a message at the same time, the message gets lost and they end up communicating nothing. It is not difficult to see that if both users start from the same initial state and use the same deterministic algorithm to communicate, then in each round, by
symmetry, they will perform the same action — either they both communicate or they both don't. Thus, there is no progress.

On the other hand, if both users use randomization, then we can show that with constant probability they will make progress in a round. To be more precise, we will assume that each user has access to random coin tosses. In particular, they can flip a fair (unbiased) coin independently and privately. This is typically referred to as an unbiased private coin (as opposed to a public coin which is shared by both users). A fair coin has probability $1/2$ of getting HEADS or TAILS. How can having such a private unbiased coin help? Here is a simple algorithm: Each user tosses his private random coin; if it comes up HEADS, then he decides to send the message, otherwise he doesn't (remains silent). See Figure 5.2. Note that this algorithm is identical to both the users except the outcome of the coin flips. Using basic probability laws, the probability that exactly one user sends a message in a round is $1/2 + 1/2 + 1/2 = 1/2$. The first term of the previous sum corresponds to the probability that user $A$ sends the message and $B$ doesn't and the second sum corresponds to the probability that user $B$ sends and user $A$ doesn’t. If we care about $A$'s message being sent, then this happens with probability $1/4$. In either case, the “success" probability, i.e., the probability of message transmission, is a constant. By repeating this algorithm a few times in every round, the success probability can be increased close to 1.

In this chapter and in later chapters we will use randomization extensively in the design of distributed algorithms. In many cases, it gives additional power to the algorithm designer, frequently leading to improved complexity bounds, and also typically simpler algorithms. In some cases, as in the above simple setting, it helps in overcoming impossibility results, i.e., without using using randomization it is not possible to solve the problem by purely deterministic means.

In randomized algorithms, we are usually interested in the average or mean or expected performance of the algorithm. The message and time complexity of a randomized algorithm is a random variable, with an associated probability distribution. We are interested in the expected value of this random variable, which is a single parameter that characterizes the distribution. In most cases, we would able to show that the random variable is concentrated around its expected value, i.e., with probability very close to 1, its value is within a small range of the expected value. This implies that “almost always" the algorithm's performance will be close to the expected value. Thus, in analysis of randomized algorithms, we usually first focus on computing the expected value.

We refer the reader to the Appendix C for a quick review of basic probability concepts and tools used throughout the book.
5.3 A Randomized LE Algorithm for Rings

We consider a randomized LE algorithm for rings, called RandLE-Ring, which is a slight modification of the deterministic BasicLE-Ring algorithm of Section 5.1.1 but it gives a significantly improved message complexity. The pseudocode is given in Algorithm 6. The main modification in the randomized algorithm, compared to the BasicLE-ring algorithm, is that instead of using IDs to break ties between nodes, this algorithm uses a randomly chosen number, called rank, to break ties between nodes; in particular, the node with the lowest rank will be elected leader. Each node chooses its rank independently of the other nodes. The rank is chosen uniformly at random from the real interval \([0, 1]\). This is an easy way to abstract the rank choosing process, which we use in many applications — it ensures that ranks are distinct with probability 1 (this is a fact from probability theory, which we just assume here for convenience). However, there is a practical difficulty in implementing it due to the fact that a real number can have infinite precision and cannot be represented precisely using a small number of bits. A typical way to overcome this is to choose a random integer in a discrete interval, namely, say between \([1, n^3]\), where \(n\) is the number of nodes in the network. This assumes that \(n\) is common knowledge among all nodes in the network, i.e., all nodes know the value of the network size. (Exercise 5.10 asks you show how one can estimate the network size.) We note that choosing the rank is the only random choice made by the nodes; all other operations are deterministic and follow the BasicLE-ring. We summarise the algorithm below. See Figure 5.3 for an illustration.

- Each node chooses a “rank”, a random (real) number in \([0, 1]\).
- Each node initiates a token that contains its rank and also its ID. The token is sent along the ring in one direction (say, in a clockwise direction).
- When a node \(v\) receives a token with a higher rank than itself the token is killed. (If the tokens have the same rank then ties are broken using IDs). Else it is allowed to progress further and \(v\) knows that it is not a leader.
- If a node receives its own token, then it becomes the leader.

5.3.1 Analysis

First we note that the correctness of the RandLE-Ring algorithm follows from the BasicLE-ring algorithm. In the RandLE-Ring algorithm, the node with the lowest rank will be elected leader (in the case when there is more than one node
Algorithm 6 RandLE-Ring: A Randomized Leader Election Algorithm for a Ring

1: \( v \) chooses a number denoted by \( \text{rank}(v) \), a random (real) number chosen uniformly in the interval \( \in [0, 1] \).
2: Round 1: Send \( (\text{rank}(v), \text{ID}(v)) \) to (clockwise) neighbor.
3: Round \( i \geq 2 \):
4: Let \( \text{rank}(u) \) be the rank of a node (i.e., it’s token) received by \( v \) at the beginning of this round.
5: if \( \text{rank}(u) < \text{rank}(v) \) or \( \text{rank}(u) = \text{rank}(v) \) and \( \text{ID}(u) < \text{ID}(v) \) then
6: Send \( (\text{rank}(u), \text{ID}(u)) \) to neighbor; \( \text{status}_v = \text{NON-ELECTED} \)
7: else if \( \text{rank}(u) = \text{rank}(v) \) and \( \text{ID}(u) = \text{ID}(v) \) then
8: \( \text{status}_v = \text{ELECTED} \)

with the lowest rank value, the one with the lowest ID will be elected as leader; others will be non-leaders.

Theorem 5.3. RandLE-Ring algorithm takes expected \( O(n \log n) \) messages and \( O(n) \) (deterministic) time.

Proof Idea: Since the algorithm is randomized, the message complexity is a random variable (however, note that the time complexity is \( O(n) \), deterministically); let’s denote this random variable by \( X \). It is not difficult to see (from the analysis of the BasicLE-ring algorithm) that this random variable will take values only in the range \([n, n^2]\). The goal is to compute the expected value of \( X \). However, computing \( E[X] \) directly from the definition is difficult, since it depends on the random choices by all the nodes which influences the actions of the intermediate nodes which determines whether a token is forwarded or not. To address this, we write \( X \) as a sum of \( X_v \)'s whose expectations are easier to compute. \( X_v \) is the random variable that denotes the number of messages generated by the token initiated by node \( v \). Thus, the main step in the proof is to compute \( E[X_v] \) and then \( E[X] \) follows easily by applying the linearity of expectation.

Proof. We compute \( E[X_v] \). \( X_v \) is equal to the distance traveled by the token initiated by node \( v \) (in the clockwise direction). Let \( x_0 \) be the random number generated by node \( v \) and \( x_i \) denotes the random number generated by the \( i \)th nearest neighbor of \( v \). The probability that \( v \)'s token gets stopped by the \( i \)th nearest neighbor is equal to the probability that \( x_i \) and \( x_0 \) are the smallest and second smallest, respectively, among \( (i + 1) \) random numbers: \( x_0, x_1, \ldots, x_i \). This probability is \( \frac{1}{i(i+1)} \) (Exercise 5.3 asks you to show this).
Thus

\[ E[X_i] = \sum_{i=1}^{n} i \cdot \frac{1}{i(i+1)} = \sum_{i=1}^{n} \frac{1}{i+1} \leq H_{n+1} = \Theta(\log n). \]

In the above, \( H_n \) is \textit{Harmonic sum}, which is defined to be the sum of the reciprocals of the first \( i \) numbers, i.e., \( H_n = \sum_{i=1}^{n} 1/i \). This is asymptotically \( \Theta(\log n) \).

To compute \( X = \sum_{v} X_v \) we use, linearity of expectation:

\[ E[X] = E[\sum_{v \in V} X_v] = \sum_{v \in V} E[X_v] = \Theta(n \log n). \]

\[ \Box \]

\section*{5.4 Leader Election in a Complete Network}

Another important special topology that is used frequently is a complete network, where all nodes have direct communication links with all other nodes. In other words, the underlying graph is a \textit{clique}. Note that the diameter of the clique is 1, the smallest possible. The clique is a dense network, which is opposite of the ring, which is a sparse network. On the other hand, the clique is also a symmetric network. A complete network can model scenario where each node can directly communicate with another node. For example, to a crude approximation, in the Internet, a node can, in principle, talk to any another node if it knows the IP address of the other node. Distributed algorithms for complete networks, are also important in that they serve as important special cases which provide lower bounds.

\subsection*{5.4.1 A Simple LE-CN Algorithm}

We first consider a simple leader election in a complete network of \( n \) nodes. This algorithm takes only one round. Each node sends its ID to all its \( n-1 \) neighbors. At the end of this round, all nodes will know the IDs of all other nodes. Thus each node can check whether it has the minimum ID among all the nodes; if so, it elects itself the leader, otherwise not. The correctness of this algorithm is immediate. It is also clear that the message complexity of this algorithm is \( \Theta(n^2) \).
5.4. LEADER ELECTION IN A COMPLETE NETWORK

5.4.2 A Randomized Algorithm

We now present a randomized algorithm, RandLE-CN, that has significantly better message complexity, and also runs in constant number of rounds. The algorithm, with high probability, solves leader election in complete networks in $O(1)$ rounds and sends no more than $O(\sqrt{n \log^{3/2} n})$ messages. We describe the main ideas of Algorithm 7 (see pseudo-code below).

Main ideas behind the algorithm. Initially, the algorithm attempts to reduce the number of leader “candidates” as far as possible, while still guaranteeing that there is at least one candidate (with high probability). Non-candidate nodes enter the NON-ELECTED state immediately, and thereafter, only reply to messages initiated by other nodes. Every node $u$ becomes a candidate with probability $\frac{2 \log n}{n}$ and selects a random rank $r_u$ chosen from some large domain. Each candidate node then randomly selects $2\lceil \sqrt{n \log n} \rceil$ other nodes as referees and informs all referees of its rank. The referees compute the maximum (say $r_w$) of all received ranks, and send a “winner” notification to the node $w$. If a candidate wins all competitions, i.e., receives “winner” notifications from all of its $\Theta(\sqrt{n \log n})$ referees, it enters the ELECTED state and becomes the leader. See Figure 5.4 for an illustration.

Algorithm 7 RandLE-CN Algorithm for a Complete Network

1: Every node $u$ decides to become a candidate with probability $\frac{2 \log n}{n}$ and generates a random rank $r_u$ from $\{1, \ldots, n^4\}$. If a node does not become a candidate, it immediately enters the NON-ELECTED state; otherwise it executes the following.

Choosing Referees:

2: Node $u$ samples (independently and uniformly at random with replacement) $2\sqrt{n \log n}$ neighbors (the referees) and sends a message $\langle u, r_u \rangle$ to each referee.

Winner Notification:

3: A referee node $v$ considers all received messages and then sends a winner notification to the node $w$ that satisfies $r_w \geq r_u$ where $r_u$ are the ranks of all tokens received by $v$.

Decision:

4: In round 2, if a node wins all competitions (by receiving $2\sqrt{n \log n}$ winner notifications), it enters the ELECTED state, otherwise it sets its state to NON-ELECTED.

In general, we will omit ceilings and floors in our analysis for convenience; this will not affect the asymptotic analysis.
Theorem 5.4. Consider a complete network $G$ of $n$ nodes and assume the CONGEST model of communication. With high probability, Algorithm 7 solves leader election in $O(1)$ rounds, while using $O(\sqrt{n} \log^{3/2} n)$ messages.

Proof Idea: We first show that there will be at least one candidate node after Step 1. We also show that there won’t be too many (more than $O(\log n)$) candidate nodes after Step 1. Both these statements hold with high probability, i.e., with probability at least $1 - 1/n$. The fact that there will be at least one candidate means that at least one leader will be elected. Not having more than $O(\log n)$ candidates leads to reduced message complexity, since only candidate nodes initiate messages. The rest of the proof is showing that exactly one candidate is elected, by showing that exactly one candidate, in particular, the one with the largest rank, will be elected. This is made possible, since each candidate node competes in about $p n$ competitions. This is the right number of competitions which guarantee that with good probability every pair of nodes will take part in at least one common competition. In particular, every node will compete with the node with the highest rank and lose.

Proof. Since all nodes enter either the elected or non-elected state after 2 rounds, at the latest, we get the runtime bound of $O(1)$.

We now argue the message complexity bound. On expectation, there are $2 \log n$ candidate nodes. By using a Chernoff bound (cf. Appendix C), there are at most $7 \log n$ candidate nodes with probability at least $1 - n^{-2}$. By the description of the algorithm, each referee node only sends messages to the candidate nodes by which it has been contacted. Since we have $O(\log n)$ candidates, the total number of messages sent is bounded by $O(\sqrt{n} \log^{3/2} n)$ with high probability.

Finally, we show that the Algorithm solves leader election with high probability. With probability $\left(1 - \frac{2 \log n}{n}\right)^n \approx \exp(-2 \log n) = n^{-2}$, no node elects itself as leader. Hence the probability that at least one node is elected as leader is at least $1 - n^{-2}$. Let $\ell$ be the node that generates the highest random rank $r_\ell$ among all candidate nodes; with high probability, $\ell$ is unique. Clearly, node $\ell$ enters the elected state, since it receives all “winner” notifications.

Now consider some other candidate node $v$. By the description of the algorithm, node $v$ chooses its referees randomly among all nodes. Therefore, the probability that an individual referee selected by $v$ is among the referees chosen by $\ell$, is $2\sqrt{n} \log n$. Since each referee is chosen independently and uniformly at random with replacement, it follows that the probability that $\ell$ and $v$ do not
choose any common referee node is at most

\[
1 - 2\sqrt{n \log n} = \exp(-4 \log n) = n^{-4},
\]

which means that with high probability, some node \(x\) serves as common referee to \(\ell\) and \(v\). By assumption, we have \(r_v < r_\ell\), which means that node \(v\) does not receive \(2[\sqrt{n \log n}]\) “winner” notifications, and thus it subsequently enters the \textsc{non-elected} state. By taking a union bound over all other candidate nodes, it follows that, with probability at least \(1 - \frac{1}{n}\), no other node except \(\ell\) wins all of its competitions, and therefore, node \(\ell\) is the only node to become a leader. □

5.5 Leader Election in a General Network

We will first present a leader election algorithm that takes \(O(D)\) time and \(O(D|E|)\) messages in a network \(G = (V, E)\) whose diameter is \(D\). Then we present a randomized algorithm that has an improved message complexity.

5.5.1 BasicLE-Gen Algorithm

We will first describe the algorithm, called BasicLE-Gen, assuming that all nodes know the diameter \(D\) — this is needed for termination detection. Later we will show how to get rid of this assumption. The algorithm BasicLE-Gen can be considered as a generalization of the BasicLE-Ring algorithm that was applied for rings. In this algorithm, each node \(v\) maintains the minimum ID that it has seen till the current round; call this as \(m(v)\). \(m(v)\) can be thought of as the current leader candidate as far as \(v\) is concerned. In the beginning (round 1), this \(m(v) = ID(v)\). In every round, a node sends its current \(m(v)\) value to all its neighbors (i.e., local broadcast) if it has not already sent this value. In other words, if \(v\) updates its current \(m(.)\) value it forwards its new \(m(.)\) value to its neighbors — call this as an update message; otherwise it sends nothing. Also, a node \(v\) when it receives a value \(m(u)\) from a neighbor \(u\), updates it current \(m(v)\) to \(m(u)\) if \(m(u) < m(v)\). After \(D\) rounds, the algorithm terminates. Algorithm 8 has the pseudocode. See Figure 5.5 for a illustration.

\textbf{Theorem 5.5.} The BasicLE-Gen algorithm is correct and has time complexity \(O(D)\) and message complexity \(O(D|E|)\).

\textbf{Proof.} Correctness. The correctness of the algorithm is easy to establish and is along the lines of the BasicLE-Ring algorithm. We first establish that the node
with the minimum ID will be elected as leader. This is because, the minimum ID will override any other ID and will be forwarded by every node. In $D$ rounds it will reach all nodes in the network. For the same reason, no other node will be elected as leader.

**Complexity Analysis.** The time complexity is trivially $D$, since the algorithm goes for $D$ rounds. In each round, every node $v$ may update its value (in the worst case) and this causes $d(v)$ update messages to be sent; thus a total of $O(|E|)$ messages are sent per round. Thus the total message complexity is $O(D|E|)$. □

**Algorithm 8** BasicLE-Gen: A Basic Leader Election Algorithm for a General Network, assuming knowledge of Diameter $D$

1: Round 1: $m(v) = ID(v)$. Send $m(v)$ to all neighbors.
2:  
3:  
4:  
5:  
6:  
7:  
8: 

**Termination detection.** If nodes do not have knowledge of $D$, then the above algorithm obviously will not work. The main problem is detecting termination. It is clear, that eventually (i.e., after $D$ rounds) the minimum ID value will spread throughout the network, but nodes won’t know whether the algorithm has finished or not. In particular, the node with the minimum ID, which will be the leader, will not know when to finally declare itself the leader. This issue can be addressed using echo messages in a way that is very similar to the one discussed in Section 3.7.2, where we discussed detecting termination of the BFS algorithm. The approach here is similar. Here, the algorithm also builds a BFS tree rooted at the node with ID $m(u)$ — call this tree $BFS(m(u))$.
5.5. LEADER ELECTION IN A GENERAL NETWORK

will be part of only one tree at any point and will drop its participation in the previous tree.) Note that \( m(u) \) is the current minimum ID learned by \( v \) (note that \( v \)'s status will be NON-ELECTED). \( v \) will send an \text{accept}(m(u)) message to \( u \); \( u \) upon receiving the \text{accept}(m(u)) from \( v \) will designate \( v \) as its child in the tree \( BFS(m(u)) \). \( v \) will then send its updated \( m(v) \) value to its neighbors. On the other hand, if \( v \) does not update its value based on the \( m(u) \) value received from a neighbor \( u \), it sends a \text{reject}(m(u)) message to \( u \).

\( v \) will check if it is a leaf node in the current BFS tree, by checking whether it receives any \text{accept} messages for an update that it sent the previous round. If \( v \) receives only reject message from all its neighbors (and no other message, e.g., a new \( m(.) \) value), then \( v \) will decide that it is a leaf for the current tree that it participates and sends an \text{echo}((m(v)) message back to its parent. If a node receives \text{echo} messages from all its children for the tree that currently participates, then it will forward its own \text{echo} message back to its parent (i.e., parent of \( v \), assuming \( v \) itself is not the root of the tree). Note that when \( v \) learns and updates a new \( m(.) \) value, then it drops waiting for \text{echo} messages from older \( m(.) \) values. (Again, a node participates in maintaining only one BFS tree at any point, the one with the smallest ID value that is currently knows.) If \( v \) itself is the root of the tree and it receives \text{echo} values from all its neighbors (for the tree \( BFS(m(v)) \)), then this means that it is the leader (sets its status to be ELECTED). At this point, the status of all nodes have been determined.

**Theorem 5.6.** The BasicLE-Gen algorithm with the termination detection as described above is correct and has time complexity \( O(D) \) and message complexity \( O(D|E|) \).

**Proof. Correctness.** As in the correctness proof of the BasicLE-Gen algorithm, we argue that the node with the minimum ID (call it \( z \)) will be elected as leader and all other nodes will be non-leaders. The value \( m(z) = ID(z) \) will be forwarded by every other node and hence all other nodes will set their status to be non-leaders. Furthermore, this means that all nodes will (finally) belong to the BFS tree rooted at \( z \). Thus the \text{echo} messages from the leaves of this tree will eventually reach \( z \) and then \( z \) will declare itself the leader.

**Complexity Analysis.** We first analyze the time complexity. The algorithm terminates when the \text{echo} messages from the tree \( BFS(ID(z)) \) reaches \( z \), the node with the minimum ID. This takes at most \( 2D \) rounds, since the depth of the tree is at most \( D \).

The message complexity is asymptotically the same as in the BasicLE-Gen algorithm because the total number of extra messages used in the termination detection version, namely, accept, reject, \text{echo} messages are asymptotically the same as the number of update messages. This is easy to see if we account
for the number of such messages generated per node in any round. A node
generates these messages in a round, only if it updates its value and sends update
messages in the previous round. For each update message, there is at most one
accept/reject message and at most one echo message.

5.5.2 A Randomized Leader Algorithm with Improved Message Com-
plexity

The message complexity of the BasicLE-Gen algorithm is high in the worst case.
Using the same approach of employing random ranks as done in the case of rings
(cf. Section 5.3), we can obtain an leader election algorithm in a general network
that has the same (deterministic) time complexity of $O(D)$, but a significantly
improved message complexity of $O(|E| \log n)$.

The randomized algorithm, called RandLE-Gen, is the same as the BasicLE-
Gen algorithm (with termination detection as well), with just one difference.
Instead of nodes using their respective IDs as ranks, they choose a random
(real) number independently and uniformly in the interval $[0, 1]$ (exactly as in
Algorithm 5). We show that choosing random ranks will lead to a substantially
improved message complexity on average.

**Theorem 5.7.** The RandLE-Gen algorithm is correct and has (deterministic) time
complexity $O(D)$ and has expected message complexity $O(|E| \min\{\log n, D\})$.

**Proof Idea:** The non-trivial part of the proof is bounding the (expected) message
complexity. The key idea is to upper bound the number of update messages
sent by some fixed node and then use linearity of expectation to bound the total
number of messages. This saves the day, since there is a non-trivial dependence
between the number of update messages sent by different nodes, e.g., the number
of messages sent by neighbors can be correlated — this makes it difficult to
bound the total number of messages directly. Instead, we focus on bounding the
expected number of messages sent by a single node and show that it is bounded
by its degree times a $O(\log n)$ factor.

**Proof. Correctness.** First, we observe that the correctness of the RandLE-Gen
immediately follows from the correctness of the BasicLE-Gen algorithm, since
the only difference is that instead of $ID(v), rank(v)$ is used as the starting value
of $m(v)$. Since ranks are chosen randomly in $[0, 1]$, with probability 1 they are
distinct. Thus the unique node with the minimum rank is elected as leader.

**Complexity analysis.** The time complexity is the same as BasicLE-Gen, and thus
is $O(D)$. 

5.6. ESTIMATING NETWORK SIZE

We next bound the expected message complexity. Let random variable $X_v$ denote the number of messages sent by node $v$ over the entire algorithm. It is clear that the total message complexity, denoted by random variable $X$, is given by $\sum_{v \in V} X_v$. We focus on computing $E[X_v]$, and then apply linearity of expectation to obtain the result.

We observe that except for the message sent by $v$ in round 1, in any subsequent round, $v$ will send a message to all its neighbors only if it updates its $m(v)$ value. The update will happen if the new $m(.)$ value received in some round is smaller than its (current) $m(v)$ value. We compute the expected number of times an update will happen at node $v$. To compute this, we consider the $m(.)$ values that $v$ receives over time. If $v$ receives more than one $m(.)$ value from its (different) neighbors in the same round, then it will update at most once (if the minimum among all the received values is less than the current $m(v)$ value).

Thus, to upper bound the number updates, it is fine to assume that $v$ sees one distinct $m(.)$ value in each round (if it does not see any new value in a round, then there is no update at all; or if it sees a value that it has already seen, then there is no update as well). Note that this is just for the sake of analysis; the algorithm itself need not operate in this way. Let the $m(.)$ values seen by $v$ starting from round 1 be $m_1, m_2, \ldots$ in that order. Each of these (distinct) values originated at some starting node (i.e, the node which first chose the rank). Hence, for the sake of analyzing the upper bound, we assume that all of these values reached $v$ (on the other hand, in the actual algorithm some of these values would have been stopped even before they reach $v$). These values are independently and randomly chosen from $[0, 1]$. Thus, we have a sequence of $n$ values, $m_1, m_2, \ldots, m_n$, each of whose values are independently and randomly chosen from $[0, 1]$. Let the random variable $Y$ denote the number of times, the minimum value is updated. By Problem C.2 (in Appendix), $E[Y] = O(\log n)$.

We can now bound the number of messages sent by $v$: $v$ sends $\deg(v)$ messages for every update. Hence $E[X_v] = O(\log n)\deg(v)$. Thus,

$$E[X] = E[\sum_{v \in V} X_v] = \sum_{v \in V} E[X_v] = \sum_{v \in V} O(\log n)\deg(v) = O(\log n) \sum_{v \in V} \deg(v) = O(|E| \log n).$$

Note that, if $2D < \log n$, then clearly, the algorithm will terminate in at most $2D$ rounds, and hence use only $O(|E|D)$ messages. \hfill \Box

5.6 Estimating Network Size

Both RandLE-Ring and RandLE-Gen requires choosing a random number in $[0, 1]$ which requires infinite precision. To avoid this, one can choose a random
number between \([1, n^3]\) (say) and it is easy to argue that, with high probability, all the \(n\) random numbers chosen (independently) by the \(n\) nodes will be unique. On the other hand, once a leader is elected (and a BFS tree is rooted at it) it is easy to compute the network size exactly: simply convergecast the count of nodes to the leader via the BFS tree. Hence, it is clear that leader election and computing network size are intimately related. For both the above randomized leader election algorithms, it is important to (at least) have an estimate of \(\log n\) (which translates into an estimate of \(n\)) to implement the random rank step. The Exercise 5.10 asks you to give an efficient randomized algorithm for this problem. Such an algorithm can be used as a preprocessing step for the above randomized leader election algorithms.

5.7 Worked Exercises

**Worked Exercise 5.1.** Give a deterministic leader election algorithm in rings that takes \(O(n)\) messages in the synchronous CONGEST model (which is obviously optimal for rings). Assume that nodes know \(n\), the number of nodes in the ring. Note that your algorithm can take an arbitrary (but finite) time.

Solution: Assume that the nodes know \(n\), the number of nodes in the network (some upper bound of it also works). Consider the following algorithm: Each node waits a certain number of rounds to start broadcasting their ID. In fact, each node starts flooding its ID at time \(ID \times n\). When a node receives an ID for the first time (including the node that sends it first), it selects that ID to be its leader. We show that this algorithm is correct and every node selects the minimum ID to be the leader.

Suppose (without loss of generality) that the IDs are \(\{1, 2, \ldots, n\}\) (if the IDs are spaced more, then this is even better for the algorithm). Then node 1, starts at time \(n\), node 2 starts at time \(2n\), node 3 starts at time \(3n\) and so on. Hence, in general, the minimum ID node will start first (at time equal to its ID) and finish its broadcast before the second minimum node starts the process. The reason is, there is a gap of at least \(n\) rounds between two consecutive starts. Therefore, after getting the minimum ID (denoted by minID), all the nodes decide upon it as the leader and the algorithm stops. Hence, the message complexity of the algorithm is \(O(n)\) (only one broadcast by the minimum ID node) and the running time is \(O(minID \cdot n + n)\) (the second term \(n\) is for completing the broadcast). So the running time is \(O(minID \cdot n)\), assuming \(minID \geq 1\).

**Worked Exercise 5.2.** Consider the following leader election algorithm for a complete network. Assume that each node arranges its neighbors in some order, say from 1 to \(n - 1\).
1. Each node independently chooses its rank, which is a random number in the interval \([0, 1]\).

2. For round \(i = 1\) to \(\log n\):
   
   (a) Each node communicates with its neighbours numbered from \(2^{i-1}\) till \(2^i - 1\) and finds their ranks.
   
   (b) If a node finds a neighbor of higher rank among the neighbours it communicates in round \(i\), it sets its status to be NON-ELECTED and stops at this round. Otherwise, it proceeds to the next round (round \(i + 1\)).

3. Set status = ELECTED.

Show that the above algorithm correctly elects a leader and takes expected \(O(n \log n)\) messages and \(O(\log n)\) (deterministic) time.

Solution:

We prove that the above protocol takes expected \(O(n \log n)\) messages and \(O(\log n)\) rounds.

It is clear that the protocol takes \(O(\log n)\) time because it takes at most \(O(\log n)\) rounds.

We now bound the expected number of messages used by the protocol.

The expected number of messages for a node is bounded by

\[
1 + \sum_{i=1}^{\log n} \left( \begin{array}{c} \text{# messages exchanged in round } i \\ \text{(Pr(not succeeding in first } i-1 \text{ rounds)}) \end{array} \right)
\]

\[
\leq 1 + \sum_{i=1}^{\log n} 2^{i+1} (1/2^{i-1}) = O(\log n)
\]

This is because, the number of messages exchanged by a node in round \(i\) at most \(2^i \times 2\) (because a node communicates with its neighbours numbered from \(2^{i-1}\) till \(2^i - 1\) (and they reply back). But the node will go to the \(i\)th round if it did not succeed in finding a higher-ranked neighbor in the first \(i-1\) rounds. In other words, it should be the highest ranked among all its \(2^{i-1} - 1\) neighbors — this probability is \(1/2^{i-1}\).

The above is the expected number of messages for a node \(v\). Linearity of expectation gives the expected total number of messages as \(O(n \log n)\).
5.8 Exercises

Exercise 5.1. Show that the time complexity of the ImprovedLE-Ring algorithm is $O(n)$.

Exercise 5.2. Give a deterministic leader election algorithm in rings that takes $O(n)$ messages in the synchronous CONGEST model (which is obviously optimal for rings). Assume that nodes don’t know $n$, the number of nodes in the ring. Note that your algorithm can take an arbitrary (but finite) time. (Hint: See Worked exercise 5.1 that gives such an algorithm, but assumes knowledge of $n$.)

Exercise 5.3. Show that in Algorithm RandLE-Ring the probability that a token generated by a node $v$ gets stopped by the $i$th nearest neighbor is equal to $\frac{1}{i(i+1)}$.

Exercise 5.4. In Algorithm RandLE-Ring, Show that choosing a rank randomly from $[1, n^2]$ (instead of from $[0, 1]$) also gives a correct leader election algorithm with the same complexity bounds.

Exercise 5.5. Give a randomized leader election algorithm in rings that takes expected $O(n)$ messages and expected $O(n)$ rounds (assume all nodes know $n$).

Exercise 5.6 (*). In a complete network, give a deterministic leader election algorithm that runs in $O(\log n)$ rounds and takes $O(n \log n)$ messages. (Hint: A natural idea is to try something similar to the deterministic leader election algorithm on the ring: reduce the number of candidates by a constant factor in every constant number of rounds. One should do this as efficiently as possible, taking on average $O(n)$ messages per round, giving $O(n \log n)$ overall.)

Exercise 5.7. Show that the complexity bounds for the BasicLE-Gen algorithm are asymptotically tight. That is, there is a network, with a given distribution of IDs such that the algorithm takes $\Omega(D)$ time and $\Omega(D|E|)$ messages for this network.

Exercise 5.8. Given an example network where the expected message complexity of the RandLE-Gen algorithm is $\Theta(|E| \log n)$.

Exercise 5.9 (*). Give a deterministic leader election algorithm in the synchronous CONGEST model that is efficient with respect to both the message complexity (i.e., close to $O(|E|)$) as well as the time complexity (i.e., close to $D$).

Exercise 5.10. In a general network, give an efficient randomized distributed algorithm to estimate the size of the network, i.e. it should output a constant factor estimate of $\log n$ with high probability. Your algorithm, with high probability, should have time complexity $O(D)$ and expected message complexity $O(|E| \min\{\log n, D\})$. 
5.8. EXERCISES

i.e., the same complexity bounds as those of RandLE-Gen. (Hint: The idea is to generate IDs that are “random”-like but not too large. Consider the following process: Each node independently tosses a fair coin till it get TAILS. It sets its ID to be the number of tosses needed to get TAILS. Show that, with high probability, the maximum ID over all nodes in the network is \( \Theta(\log n) \).)

**Exercise 5.11.** Use the ideas in Exercise 5.10 to design and analyze a leader election algorithm in a general network, where nodes don’t have any knowledge of \( n \), the size of the network.
CHAPTER 5. LEADER ELECTION

Figure 5.1: Illustrating BasicLE-Ring algorithm.
Figure 5.2: Symmetry breaking using randomization.
Figure 5.3: Illustrating RandLE-Ring algorithm.
Here, A, B, C, D, E, F are ID’s and 10, 15, 20, 25, 30, 35, 40 are corresponding ranks.

let A, C and E are candidates and B, D, F are referees for all of them.

A, C, E sends message to all referees

winner messages sent by referee’s

A wins all competitions

State: Elected

Figure 5.4: Illustrating RandLE-CN algorithm.
Figure 5.5: Illustrating BasicLE-Gen algorithm.
Chapter 6

Local Symmetry Breaking

In the last chapter, we focused on leader election, which can be considered as a “global” symmetry breaking problem since the goal is to break symmetry among all the nodes in the network in order to elect a unique leader. In contrast, in this chapter, we look at “local” symmetry breaking, where the goal is to break symmetry among nodes that are quite close, typically neighbors. A fundamental problem in this category is the Maximal Independent Set (MIS) problem which is stated as follows: Given a graph $G = (V, E)$, find a subset $S \subseteq V$ of nodes such that $S$ is independent (i.e., no two nodes in $S$ are neighbors in $G$) and $S$ is maximal (i.e., no other nodes can be added to $S$ and still keep it independent). MIS is a “local” problem in the sense that given a subset $S \subseteq V$, verifying whether $S$ forms a maximal independent can be done as follows: each node checks with its neighbors and decides whether the MIS property holds in its neighborhood. However, as we will see, computing a MIS in a distributed network is harder. In particular, we will present a MIS algorithm that takes $O(\log n)$ rounds (where $n$ is the number of nodes in the network) with high probability. Another way to view this algorithm is as follows: To compute a MIS, each node (with high probability) needs only information about its $O(\log n)$-neighborhood, i.e., nodes within distance $O(\log n)$. While this distance is not “fully local” (i.e., not just restricted to looking at neighbors alone), but still relatively small compared to the network diameter (which can be as large as $n$). A fundamental open question in distributed computing is to fully characterize the locality of MIS. As we shall see in a later chapter, we can show a highly non-trivial locality lower bound of $\Omega(\log^* n)$ for MIS, i.e., in the worst case, some node has to look at information from nodes that are $\Omega(\log^* n)$ away.

$^1\log^* n$ is the iterated logarithm of $n$ — the number of times one has to repeatedly take logarithmic function to reduce $n$ to 1.
In this chapter, besides MIS, we will focus on another fundamental local
symmetry breaking problem, namely, coloring. Both are related to each other in
some way, yet have their own characteristics. We will also study a distributed
optimization problem, namely finding small-sized dominating sets. We will
show fast (i.e., running in polylog(n) rounds) distributed algorithms for all
these problems. A key feature of all the algorithms in this chapter is the use of
randomness — all the algorithms are randomized. This is not a coincidence —
currently the best known algorithms for these problems are randomized and are
significantly faster than the best known deterministic counterparts.

Throughout this chapter, we will consider the synchronous LOCAL model,
although all the algorithms (for MIS, coloring, and dominating set) will work
seamlessly in the CONGEST model as well. As usual, we consider a network of n
nodes, represented as an undirected, connected (not necessarily complete) graph
\( G = (V, E) \). We assume that all nodes have unique identifiers; the identifiers
can be arbitrary numbers which can be represented by \( O(\log n) \) bits. As in the
previous chapter, we will assume that all nodes are awake initially and start
executing the algorithm simultaneously, i.e., the simultaneous wakeup model.

6.1 Maximal Independent Set (MIS)

Given an arbitrary network we want to find a maximal independent set of nodes.
We formally define a MIS below. Figure 6.1 gives an example.

**Definition 6.1.1.** Given a graph, a subset of nodes is called an independent set
if it contains no pair of neighboring nodes, and it is maximal if it cannot be
increased to form a larger independent set by the addition of other nodes.

MIS is a basic primitive that arises in distributed algorithms. It has many
applications, e.g., resource allocation, dominating set construction etc. For exam-
ple, in a wireless network, neighboring nodes may not be able to communicate
properly due to signal interference. One way to solve this problem is by finding
a MIS and allowing nodes that belong to the MIS to communicate at the same
time. Another useful property of MIS is that it is also a dominating set. Given a
graph \( G = (V, E) \), a dominating set \( D \subseteq V \) is a subset of nodes such that every
vertex in \( V \) either belongs to \( D \) or has a neighbor in \( D \). It is easy to see that a MIS
is also a dominating set; in fact it is a minimal dominating set (MDS). A MDS can
be used in various applications. For example, a dominating set can be used as a
network backbone for routing — it is enough to find routes between the nodes
in the dominating set; any other node can route by sending it to its dominator
first (if there is more than one dominator, a specific one can be chosen).
6.1. MAXIMAL INDEPENDENT SET (MIS)

(a) independent sets \{A, E\}, \{E, C\}, \{D, E\}, \{A, E, C\} etc
(b) \{A, E, C\} is Maximal Independent Set. Because we can not add more nodes in this set.

Figure 6.1: Maximal Independent Set.

Fast Distributed MIS Algorithms  Given an arbitrary network \(G = (V, E)\), the goal is to design a distributed algorithm which will output a MIS as follows: every node will know whether it is in the MIS or not.

Let \(\Gamma(v)\) be the set of vertices in \(V\) that are adjacent to \(v\). Let \(N(v)\) denote the closed neighborhood of \(v\), i.e., the set consisting of \(v\) and \(\Gamma(v)\).

The goal is to design fast distributed algorithms, i.e., algorithms running in \(O(\log n)\) (or \(O(\text{polylog}(n))\)) rounds. The reason we call these algorithms “fast” is because their running time can be significantly smaller than the diameter \(D\) of the network. Note that \(D\) is an obvious time upper bound for any graph problem in the LOCAL model, since one can collect all the network information (including topology) at a single node and locally (within node) compute the solution.

Intuition behind the Fast MIS Algorithms  The algorithm proceeds in rounds; in every round it finds an independent set \(S\), and adds \(S\) to \(I\) (initially \(I\) is empty) and deletes \(S \cup \Gamma(S)\) from the graph. Note that it is very easy to implement this algorithm in a linear (i.e., \(O(n)\)) rounds (see Exercise 6.3). This is essentially the same as finding a MIS in the sequential setting by iterating the following starting with the input graph \(G\): Choose an arbitrary node \(v\) in \(G\) and include it in the MIS, and delete \(v\) and its neighbors in \(G\) to get a new graph \(G'\); repeat the same in \(G'\) till \(G'\) becomes empty.

We give some intuition in designing an algorithm that runs in \(O(\log n)\) rounds, which is significantly faster than a linear time algorithm. To get a fast distributed algorithm, an obvious strategy is to include as many nodes in the MIS in a round as possible. Consider a node \(v\) whose degree is denoted by \(d(v)\). Clearly either...
or at least one of its neighbors should be in the MIS. (In particular, if \( v \) is in the MIS, that precludes its neighbors to be in the MIS.) Thus, we have a basic symmetry breaking problem between \( v \) and its neighbors. As discussed earlier (see Chapter 5 in regard to the toy problem of channel usage), the symmetry can be broken by using randomization. One reasonable way is for \( v \) to choose itself with probability \( \frac{1}{d(v)+1} \). Why? Assume, for simplicity, that all nodes in the graph have the same (or almost same) degree. Then the above probability ensures that on average there will be one node chosen among \( v \) and its neighbors (this follows simply by linearity of expectation) which is correct “on average.”

But due to randomness two situations are possible: (1) Two neighboring nodes can be chosen; (2) None of the nodes are chosen in a closed neighborhood of a node. To deal with the first situation, the algorithm does a (deterministic) tie breaking rule if two neighboring nodes are chosen: for example, one can break ties based on degrees (or if the degrees or the same, then based on IDs).

There is an alternate way of probabilistic tie breaking that avoids the above situation. Each node chooses a random rank (similar to randomized leader election algorithms!); the node with the lowest rank goes into the independent set. The random rank is chosen from a large enough set (say \([1, n^3]\)), then the probability that two nodes choosing the same rank is negligible (one can also implement this random choosing without knowledge of \( n \) — this is explored in Exercise 6.1). To deal with the second situation, the algorithm is repeated many times till the status of all nodes are determined. We will show that, with high probability, the number of repetitions will be small, i.e., logarithmic in \( n \).

Next we present two algorithms for MIS based on the two different symmetry breaking ideas discussed above. Both these ideas are important as similar themes show up in other applications as well such as coloring and dominating set.

### 6.1.1 MIS Algorithm 1

The first MIS algorithm that we present is based on the first symmetry breaking strategy discussed above. Each node chooses itself to be in the MIS with probability \( 1/(2d(v)) \) (this is slightly smaller than \( 1/(d(v) + 1) \) — this “slack” is useful for analysis). To handle the scenario, that two neighboring nodes choose themselves, ties are broken based on the degrees: the tie is broken in favor of the higher degree node. This is slightly counter-intuitive, but makes sense: if a higher degree node is favored in tie breaking, it will eliminate more (of its) neighbors. See Figure 6.2 for an illustration.
Algorithm 9 MIS Algorithm 1: Code for node $v$

1: $status(v) = \text{undecided}$
2: while $status(v) = \text{undecided}$ do
3: \hspace{1em} if $d(v) = 0$ then
4: \hspace{2em} $status(v) = \text{yes}$ // belongs to MIS
5: \hspace{1em} else
6: \hspace{2em} $v$ marks itself with probability $1/(2d(v))$ // marking step
7: \hspace{2em} $v$ notifies neighbors that it is marked and also sends its (current) degree
8: \hspace{1em} if $v$ receives a message from a marked neighbor of higher degree (or equal degree but higher ID) then
9: \hspace{2em} $v$ unmarks itself // tie-breaking step
10: \hspace{1em} if $v$ is (still) marked then
11: \hspace{2em} $status(v) = \text{yes}$
12: \hspace{2em} $v$ notifies its status to its neighbors
13: \hspace{2em} if $v$ receives a message from a neighbor that is in MIS then
14: \hspace{2em} $v$ deletes itself and all its edges from the network
15: \hspace{1em} $status(v) = \text{no}$ // does not belong to MIS

Analysis of MIS Algorithm 1

A simpler analysis: $O(\log n \log \Delta)$ rounds

We will first present a simpler analysis of the algorithm that shows a weaker time bound of $O(\log n \log \Delta)$ rounds, where $\Delta$ is the maximum node degree. Later we will modify this analysis to show a stronger $O(\log n)$ bound.

Theorem 6.1 (Weaker bound). MIS Algorithm 1 runs in $O(\log n \log \Delta)$ rounds with high probability.

The main idea of the analysis is as follows. We divide the algorithm into phases (for the sake of analysis): in the first phase, we will consider nodes having degree between $[\Delta, \Delta/2)$; in phase $i$ the nodes considered will have degree between $[\Delta/2^{i-1}, \Delta/2^i)$. Thus there will be $O(\log \Delta)$ phases. At the end of phase $i$, the status of all nodes of degree higher than $\Delta/2^i$ would have been decided. We will show that each phase lasts for $O(\log n)$ rounds with high probability. The following is the key lemma.

Lemma 6.1. Consider a node, say $v$, in phase $i \leq i \leq \log \Delta$. We will show that, with constant probability, one of the following two events will happen: (1) The status of $v$ will be determined (and hence will take no further part in the algorithm); (2) The degree of $v$ will drop below $\Delta/2^i$. 
Proof. Consider phase 1. In this phase, we only focus on nodes with degree $[\Delta, \Delta/2]$; let $v$ be one such node. We lower bound the probability that the status of $v$ will be determined in one round. This can be done in two ways in a round: (1) $v$ enters MIS or (2) a neighbor of $v$ enters MIS.

We lower bound the probability that a neighbor of $v$ enters MIS as follows in two steps: (1) A neighbor of $v$, say $w$, marks itself; and (2) At least one of $v$'s marked neighbors remains marked after the tie-breaking step (line number 9 of Algorithm 9).

The probability that step (1) does not happen is at most 
\[ \left(1 - \frac{1}{2\Delta}\right)^{\Delta/2} \leq e^{-1/4}. \]
This is because all the neighbors of $v$ have degree at most $\Delta$ and $v$'s degree $d(v)$ is at least $\Delta/2$. Hence the probability that step (1) happens is at least $1 - e^{-1/4}$.

We next bound the probability of step (2), given step (1) has happened. Among all the marked neighbors of $v$ consider the one that has the highest degree, call it $w$ (choose the one with the highest ID if there is more than one such node). (See Figure 6.3.) The probability of $w$, remaining marked is at most the probability that none of its higher (or equal) degree neighbors are marked. Crucially, it is enough to focus on neighbors of $w$ that are not in $\Gamma(v)$. This is because, $w$ is the highest degree node in $\Gamma(v)$ (or it has the highest ID if there is another marked node in $\Gamma(v)$). Note that, we also do not consider the event that $v$ itself is of higher degree than $w$ and is marked; this event will lead to $v$'s deletion anyway.

Thus we can bound the probability of the following event independent of the nodes in $N(v)$: the probability that at least one of the neighbors of $w$ (excluding those in $N(v)$) is marked is at most
\[ \sum_{u \in \Gamma(w)} \frac{1}{2d(w)} \leq \frac{1}{2}. \]
Thus the probability that none of its neighbors are marked is at least $1/2$.

We next lower bound the probability that both steps (1) and (2) will happen. By the above discussion, due to the independence of the two steps, the probability is at least $(1 - e^{1/4})(1/2)$. Let's denote this constant probability by $\beta$.

After $O(\log n)$ iterations of the While loop, with high probability either $v$ is deleted or its degree will go below $\Delta/2$, i.e., it will move out of phase 1. By a straightforward union bound, the above statement will hold for all nodes considered in phase 1. Thus in $O(\log n)$ rounds, with high probability, all nodes with degree in $[\Delta, \Delta/2)$ will be deleted (i.e., their status will be determined) or
6.1. **MAXIMAL INDEPENDENT SET (MIS)**

Their degrees will be reduced below $\Delta/2$. We can then have a similar argument for phase 2, and in general, phase $i$. By applying another union bound over all the $O(\log \Delta)$ phases, the status of all nodes will be determined with high probability in $O(\log \Delta \log n)$ rounds.

**Better analysis: An $O(\log n)$ round algorithm** We can improve the analysis by considering all vertices (regardless of their degrees) at the same time, instead of categorizing them by degrees as done in the simpler analysis earlier. But the main idea is similar. We consider a vertex that has many neighbors whose degrees are smaller than itself (this is similar to the situation in a phase in the previous analysis). We show that the probability that this vertex will be deleted is at least a constant. Then, we will show, that this will imply, on the average, a constant fraction of the edges will be deleted in each iteration of the While loop.

**Definition 6.1.2.** A vertex $v$ supercedes another vertex $u$ if either: (1) $d(v) > d(u)$ or (2) $d(v) = d(u)$ and $I_D(v) > I_D(u)$. A vertex $v \in V$ is good if it supercedes at least one-third fraction of its neighbors; otherwise the vertex is bad. An edge is good if at least one of its endpoints is a good vertex, and it is bad if both endpoints are bad vertices.

We show that a good vertex is likely to have one of its lower degree neighbors in the MIS and thereby deleted from $V$.

**Lemma 6.2.** Let $v \in V$ be a good vertex with degree $d(v) > 0$. Then, the probability that some vertex $w \in \Gamma(v)$ gets marked is at least $1 - e^{-1/6}$.

**Proof.** Each vertex $w \in \Gamma(v)$ is marked independently with probability $1/(2d(w))$. The probability that none of the neighbors of $v$ gets marked is at most

$$\left(1 - \frac{1}{2d(v)}\right)^{d(v)/3} \leq e^{-1/6}.$$

\[\Box\]

The proofs of the next two lemmas are very similar to the argument in the proof of Lemma 6.1 and are left as exercises.

**Lemma 6.3.** Consider an iteration of the While loop of Algorithm 9. Let $v$ be a good vertex and let $w$ be a marked neighbor of $v$ that has the highest degree (if there is a tie, then $w$ is the one with the highest ID). Then $w$ is selected to be in the MIS with probability at least $1/2$. 

**Lemma 6.4.** The probability that the status of a good vertex is determined in an iteration of the While loop is at least \((1 - e^{-1/6})/2\).

We next show a crucial fact that holds in any graph (deterministically).

**Lemma 6.5.** In a graph \(G = (V, E)\) the number of good edges is at least \(|E|/2\).

**Proof.** Direct the edges in \(E\) from the lower degree end-point to the higher degree end-point breaking ties according to the ID (as done in the algorithm).

We will show a mapping that assigns each bad edge to two unique edges. This will imply that the number of bad edges is at most \(|E|/2\), proving the lemma. The mapping is as follows. Consider a bad vertex \(v\). At least two-thirds of its neighbors supercede \(v\), and all these edges will be directed out of \(v\); at most one-third will be directed into \(v\). Consider a bad edge \(e\) directed into \(v\). Map \(e\) to two distinct outgoing edges of \(v\). Since there are twice as many outgoing edges as incoming edges, one can uniquely map every incoming (bad) edge to two unique outgoing edges. This is the required mapping. \(\square\)

**Lemma 6.6.** The expected number of edges deleted in one iteration of the algorithm is at least \(((1 - e^{-1/6})/4)|E|\).

**Proof.** By lemma \(6.4\), a good edge is deleted with probability at least \((1 - e^{-1/6})/2\), since at least one of its endpoints is good. Since the number of good edges is at least \(|E|/2\), the claim follows by linearity of expectation. \(\square\)

**Lemma 6.7.** MIS Algorithm 1 finishes in \(O(\log n)\) rounds with high probability.

**Proof.** Let \(\alpha = 1 - (1 - e^{-1/6})/4\). Let \(X_i\) be the number of edges remaining after \(i\) iterations. We have \(X_0 = |E|\). By Lemma \(6.6\) for \(i \geq 1\), \(E[X_i|X_{i-1}] \leq \alpha X_{i-1}\).

By taking expectations on both sides (see Appendix ??),

\[E[E[X_i|X_{i-1}]] = E[X_i] \leq \alpha E[X_{i-1}].\]

Hence

\[E[X_i] \leq (\alpha)^i E[X_0] = (\alpha)^i |E|\]

Choosing \(i = f = 4 \log_{1/\alpha} n\), we obtain \(E[X_f] \leq |E|/n^4 \leq 1/n^2\), since \(|E| \leq n^2\).

Using Markov’s inequality, we bound the number of edges remaining after \(f\) iterations:

\[\Pr(X_f \geq 1) \leq E[X_f] \leq 1/n^2.\]

\(\square\)
6.1.2 Another distributed algorithm for MIS: MIS Algorithm 2

Consider the following alternate randomized distributed algorithm for finding a maximal independent set that is based on the second symmetry breaking strategy discussed in the beginning of Section 6.1. We will show that this algorithm runs in $O(\log n)$ rounds with high probability.

**Algorithm 10 MIS Algorithm 2: Code for node $v$**

1: $status(v) = \text{undecided}$
2: while $status(v) = \text{undecided}$ do
3: if $d(v) = 0$ then
4: $status(v) = \text{yes} // \text{belongs to MIS}$
5: else
6: mark $v$
7: Independently and uniformly choose a random (real) number from the interval $[0, 1]$. Call this the rank of $v$
8: $v$ notifies its rank to its neighbors
9: if $v$ receives a message from a lower ranked neighbor then
10: $v$ unmarks itself // tie-breaking step
11: if $v$ is (still) marked then
12: $status(v) = \text{yes} // v$ is the lowest ranked among its neighbors
13: $v$ notifies its status to its neighbors
14: if $v$ receives a message from a neighbor that is in MIS then
15: $v$ deletes itself and all its edges from the network
16: $status(v) = \text{no} // \text{not belongs to MIS}$

**Analysis of MIS Algorithm 2**

**Lemma 6.8.** In one iteration of the While loop of the MIS Algorithm 2, the expected number of edges deleted is at least half the number of edges in the (current) graph.

**Proof.** Let $F$ be the set of edges at the beginning of an iteration (in the beginning of the algorithm $F = E$, where $E$ is the set of edges of the graph). We show that the expected number of edges deleted in the iteration is at least $|F|/2$. For the sake of analysis, replace each undirected edge $(u, v)$ by two directed edges $(u \to v)$ and $(v \to u)$. Let $N(u)$ denote the set of neighbors of $u$ (in the undirected graph) and $u$ itself. Call a node $u$ as "eligible" with respect to $v$ if $u$ is the smallest ranked node among $N(u)$ and $N(v)$. Node that if $u$ is eligible it will be deleted, since it is the lowest ranked node in $N(u)$ and hence will get into
the MIS. It is clear that:

\[ \Pr(u \text{ is eligible w.r.t. } v) \geq \frac{1}{d(u) + d(v)}. \]

Let random variable \( X(u \rightarrow v) \) denote the number of directed outgoing edges incident to \( v \) that get deleted when \( u \) is eligible with respect to \( v \). We note that \( X(u \rightarrow v) \geq d(v) \) if \( u \) is eligible with respect to \( v \). Note that we deliberately undercount the number of directed outgoing edges deleted — in fact, \( d(u) + d(v) \) edges are deleted when \( u \) is deleted. But this is to avoid overcounting as explained below.

Let r.v. \( X \) denote the total number of directed edges deleted in the iteration. Then

\[
X \geq \sum_{(u \rightarrow v), (v \rightarrow u) \in F} X(u \rightarrow v) + X(v \rightarrow u).
\]

Note that when \( u \) deletes outgoing edges of \( v \) because of the reason that \( u \) is eligible w.r.t. \( v \), no other neighbor of \( v \) can be simultaneously eligible. Thus each deleted (directed outgoing) edge is counted at most once.

By linearity of expectation, the expected total number of (directed) edges deleted is

\[
E[X] \geq \sum_{(u \rightarrow v), (v \rightarrow u) \in E} E[X(u \rightarrow v)] + E[X(v \rightarrow u)]
\]

\[
\geq \sum_{(u \rightarrow v), (v \rightarrow u) \in E} \frac{d(v)}{d(u) + d(v)} + \frac{d(u)}{d(u) + d(v)}
\]

\[
= \sum_{(u \rightarrow v), (v \rightarrow u) \in F} \frac{d(v) + d(u)}{d(u) + d(v)} = |F|.
\]

Since we counted directed edges, the actual number of (undirected) edges deleted is \(|F|/2\).

**Lemma 6.9.** The algorithm terminates in \( O(\log n) \) iterations with high probability, i.e., with probability at least \( 1 - 1/n \).

**Proof.** The proof is similar to the proof of Lemma 6.7. Using conditional expectation, the expected number of edges remaining after \( k \) iterations is at most \(|E|/2^k\). Plugging \( k = 4 \log n \), the number of edges remaining will be at most \(|E|/n^4 \leq 1/n^2 \), since \(|E| \leq n^2 \). By Markov’s inequality the probability that at least one edge will remain after \( 4 \log n \) iteration is at most \( 1/n^2 \).
Choosing a random number Since we choose a random number in $[0, 1]$, every node gets a unique rank with probability 1 (because of infinite precision), so the algorithm will correctly produce an MIS. However, it is not crucial that all ranks are distinct. If they are not, then there is a small probability that the algorithm will be incorrect.

Thus, one can simply choose a random number from the interval, say, $[1, n^4]$, so that the probability that any two nodes will have the same rank in any one iteration is at most $(\frac{n}{2})1/n^4 = O(1/n^2)$. Thus, with probability at least $1 - O(1/n)$, during the entire course of the algorithm, all ranks will be distinct (why?).

Thus, we will have a Monte-Carlo algorithm that is correct with high probability. If we want a Las Vegas algorithm, we can simply rerun the algorithm if we find that two neighbors have been included in the independent set. This will still guarantee an expected running time of $O(\log n)$. Hence we can assume that all ranks are distinct (thus nodes have been assigned a random ordering of ranks).

6.2 Coloring

In this section, we consider another fundamental local symmetry breaking problem, namely coloring a graph. In particular, we consider the problem of coloring a given graph using $\Delta + 1$ colors, where $\Delta$ is the maximum degree of the graph. It is easy to show that every graph can be colored in $\Delta + 1$ colors (see Exercise [6.4]). In fact, it is easy to accomplish this using a centralized algorithm in linear time. Here we will show how to achieve the same using a distributed algorithm that runs in $O(\log n)$ rounds.

Given an undirected graph $G$, a vertex-coloring of $G$ is simply assigning colors to vertices. We are interested in a legal vertex-coloring, i.e., no two neighboring vertices should be assigned the same color (in other words, if two vertices are connected by an edge they should be assigned different colors).

6.2.1 A ($\Delta + 1$)-Coloring Algorithm

Consider the following distributed algorithm for computing a legal vertex-coloring of a graph.

Every vertex $u$ in $G$ initially has a full list of colors $L_u = \{1, 2, \ldots, d_u + 1\}$, where $d_u$ is the degree of $u$. (Each number denotes a different color and there are $\max_x \{d_x + 1\} \leq \Delta + 1$ colors to begin with.) The algorithm proceeds in iterations (Algorithm 11 gives the pseudocode). Initially all vertices are uncolored and assume that all vertices are asleep. Each iteration consists of the following steps:
1. Every vertex not yet colored wakes up with probability 1/2.

2. Every vertex that woke up picks a tentative color uniformly at random from its own list of colors.

3. If $t_u$ is a tentative color picked by $u$, and no neighbor of $u$ picked $t_u$, then $u$ colors itself with $t_u$.

4. The color list of each uncolored vertex in the graph is updated by removing all colors successfully used by neighbors.

5. All colored vertices are deleted from the graph (they don’t take further part in the algorithm).

6. All uncolored vertices go back to sleep.

**Algorithm 11** Coloring Algorithm: Code for node $v$

1: $L_v = \{1, 2, \ldots, d_v + 1\}$, where $d_v$ is the degree of $v$ // $L_v$ is the initial color list of $v$
2: $\text{status}(v) = \text{uncolored}$
3: $\text{wakeup}(v) = \text{false}$ // initially $v$ is asleep
4: while $\text{status}(v) = \text{uncolored}$ do
5: With probability 1/2, set $\text{wakeup}(v) = \text{true}$
6: if $\text{wakeup}(v) = \text{true}$ then
7: Choose a tentative color $t_v$ independently and uniformly at random from $L_v$
8: Announce tentative color $t_v$ to all neighbors
9: if no neighbor of $v$ picked $t_v$ as its tentative color then
10: $\text{color}(v) = t_v$ // $v$ colors itself with color $t_v$
11: Announce color $t_v$ to all neighbors
12: $\text{status}(v) = \text{colored}$
13: else
14: $\text{wakeup}(v) = \text{false}$ // $v$ goes back to sleep
15: if a neighbor has colored itself with color $c$ then
16: $L_v = L_v - \{c\}$

**Analysis** We show the correctness and analyze the running time of the algorithm.

**Lemma 6.10.** The algorithm (if it terminates) computes a legal vertex-coloring of the graph.
6.2. COLORING

Proof. Note that in any step, a vertex $u$ always chooses a color that is not chosen by any of its neighbors (in this step). Since colors chosen by neighbors that were colored in previous steps are deleted from $u$’s list, there is no conflict. Since every vertex starts with $d_u + 1$ colors, each uncolored vertex is left with at least one color to choose from at any step. Hence the coloring is legal.

Lemma 6.11. In every iteration, each uncolored vertex successfully colors itself with probability at least $\frac{1}{4}$.

Proof. Consider an uncolored vertex $u$ at a iteration $t$. Let $L^t_u$ denote the list of colors that $u$ has at the beginning of this round. Denote the (current) degree of $u$ as $d^t_u$ and the (current) set of neighbors of $u$ at $N^t_u$. We will show that with constant probability $u$ will color itself successfully in iteration $t$.

Let the event $E_u$ denote that vertex $u$ colors itself successfully. Let $\overline{E}_u$ denote the complement of the event $E_u$. Let the event $E_u(c)$ denote that $u$ chooses color $c$ (in this round).

Since a node has to wake up to color itself, we have,

$$
\Pr(E_u) = \Pr(E_u|W_u) \Pr(W_u) = \frac{1}{2} \Pr(E_u|W_u) = \frac{1}{2} (1 - \Pr(\overline{E}_u|W_u)). \quad (6.1)
$$

We upper bound $\Pr(\overline{E}_u|W_u)$. This event will happen if any one of $u$’s neighbors wakes up and also chooses the same color that $u$ chooses. Hence,

$$
\Pr(\overline{E}_u|W_u) \leq \sum_{v \in N^t_u, c \in L^t_u} \Pr(E_u(c) \land W_v \land E_v(c)|W_u)
$$

$$
= \sum_{v \in N^t_u, c \in L^t_u} \Pr(E_u(c) \land E_v(c)|W_u, W_v) \Pr(W_v)
$$

$$
= \frac{1}{2} \sum_{v \in N^t_u, c \in L^t_u} \Pr(E_u(c) \land E_v(c)|W_u, W_v)
$$

To bound the above probability, fix a particular neighbor $v$. Exercise 6.6 asks you to show that the probability that $u$ and $v$ choose the same color is at most $\frac{1}{|L^t_u|} = \frac{1}{(d^t_u + 1)} < \frac{1}{d^t_u}$. Hence,

$$
\Pr(\overline{E}_u|W_u) < \frac{1}{2} \sum_{v \in N^t_u} \frac{1}{d^t_u} = \frac{1}{2}.
$$

Hence, by equation 6.1, $\Pr(E_u) \geq \frac{1}{4}$. ☐
Lemma 6.12. Within $O(\log n)$ rounds the graph will be legally colored with high probability (i.e., with probability at least $1 - 1/n$), where $n$ is the number of vertices in $G$.

Proof. The probability that a vertex is not colored after $16 \ln n$ rounds is at most $1/n^2$. By union bound, the probability that no vertex remains uncolored after so many rounds is at most $1/n$. 

6.3 Deterministic Symmetry Breaking

We can solve local symmetry breaking problems faster than the $O(\log n)$-round algorithms we have seen so far for some special classes of graphs. In this section, we will present $O(\log^* n)$-round algorithms for coloring and MIS in bounded degree graphs and for other graphs such as rooted trees. What is perhaps more surprising, besides the super-fast running time of $O(\log^* n)$ is that these algorithms are purely deterministic. They use a simple, yet powerful idea of breaking symmetry that is best illustrated in rooted trees.

6.3.1 Coloring in Rooted Trees

We are given an arbitrary rooted tree, i.e., each node knows the identity of its parent (if any) and children (if any). Let $n$ be the number of nodes. We present an algorithm that colors the rooted tree using 6 colors. A typical paradigm used in many coloring algorithms is as follows (although this paradigm is not used in the coloring algorithm of Section 6.2.1). Start with a legal coloring (but with a large number of colors) and in each round reduce the total number of colors used while maintaining legality. An easy coloring to start with is assigning to each node its (unique) ID. Thus we start with a legal coloring of $n$ colors. The symmetry breaking technique of the algorithm reduces the number of colors to (essentially) the logarithm of the existing number of colors in each round; thus in $O(\log^* n)$ rounds the number of colors is reduced to a constant. Let $<c> = c_\ell, c_{\ell-1}, \ldots, c_1, c_0$ denote the bit representation of color $c$ ($c_0$ is the least significant bit). Initially, the number of bits needed to represent a color — denoted as $|<c>|$, the size of the color — is $O(\log n)$ (i.e., $\ell = O(\log n)$), since, that is the length of a node ID. The key deterministic symmetry breaking idea used is as follows: Each node (except the root) compares its color, say $<c>$, with the color of its parent, say $<c'>$. Let $i$ be the index of the least significant bit where $<c>$ differs from $<c'>$. (For example, if $<c> = 100010$ and $<c'> = 100110$, then $i = 2$, because the 0th bit and the 1th bit of the two colors are the same, but they differ in the 2nd bit.) Then the node changes its color to...
6.3. DETERMINISTIC SYMMETRY BREAKING

\[ i \rightarrow c_i, \] i.e., the new color is a concatenation of the bit representation of \( i \) and the \( i \)th bit of \( c \). Each node then announces its color to its children and the process is continued in the next round. The process terminates for a node when there is no further reduction in its color size. Algorithm 12 gives the pseudocode of the coloring algorithm. We argue its property and run time in the following lemma.

**Theorem 6.2.** Algorithm 12 on a rooted tree produces a legal 6-coloring in \( O(\log^* n) \) rounds.

**Proof.** We first argue that each iteration of the WHILE loop maintains a legal coloring.

At the beginning the coloring is obviously valid. Hence, assuming that the coloring is valid at the beginning of an iteration of the WHILE loop, we show that it remains valid at the end of the iteration. Consider any two adjacent nodes \( u \) and \( v \), and let \( u \) be the parent of \( v \). For the moment, we will assume that \( u \) is not the root. In an iteration, if \( u \) and \( v \) choose different indices (in Line 9 of Algorithm 12) where they differ with respect to their respective parents, then the new colors of \( u \) and \( v \) are obviously different. On the other hand, if they both choose the same index (say \( i \)), then that means their \( i \)th bits are different; hence the new colors will be different again (because the \( i \)th bit is part of the color as well). If \( u \) is a root, then a similar argument holds.

We now argue that each iteration reduces the size of the color of each node by (essentially) a logarithmic factor. This is because if \( \ell \) is the size of the color at the beginning of the iteration, then the size of the color at the end of the iteration is at most \( \lceil \log \ell \rceil + 1 \), since it is a concatenation of the representation of an index of the color (which has at most \( \lceil \log \ell \rceil \) bits plus the concatenation of 1 bit).

Thus after \( O(\log^* n) \) iterations the number of colors will be reduced to a constant. The algorithm continues till the size of the color does not reduce any further. It is easy to check that this means that the node ends up with the following six colors (represented as bits): 00, 01, 10, 11, 100, 101.

**From 6-coloring to 3-coloring**

We show how the 6-coloring produced by Algorithm 12 can be converted into a 3-coloring. This can be accomplished by using the following strategy of “shifting down” colors in the tree. This strategy is as follows: Each node (except the root) adopts the color of its parent. The root chooses a (any) color different from its current. This strategy clearly ensures legality. But it gives an additional property:
Algorithm 12 \(O(\log^* n)\) Coloring Algorithm for Rooted Trees (code for node \(v\)).

1: \(c(v) = ID(v)\) \//Initialize color of \(v\) to be its ID.
2: send color to all children (if any)
3: \(flag = "true"\)
4: while \(flag == "true"\) do
5: \(oldsize = |<c(v)>|\) \// Number of bits in the bit representation of \(c(v)\)
6: if \(v == root\) then
7: \(i = 0\) \// picks index as 0
8: else
9: Let \(i\) be the first (least significant) index where \(<c_\text{parent}(v)>\) differs from \(<c_v>\)
10: \(<c(v)> = <<i > c_i(v)>\) \// Set the new color of \(v\) as the concatenation of the bit representation of \(i\) and the \(i\)th bit of \(c(v)\).
11: \(v\) notifies its color to all its children
12: if \(oldsize == |<c(v)>|\) then
13: \(flag = "false"\) \// Finish the algorithm

All the children will have the same color. This means that for every node, its neighbors will be using at most two colors.

Using the shifting-down strategy we now eliminate colors 4, 5, and 6 if present in the coloring as follows. Each color is eliminated in 2 rounds. Let’s focus on color 4. A shifting-down of colors is performed. Then each node having color 4 looks at its neighbors and chooses a “free” color from the set \(\{1, 2, 3\}\). This is always possible, since after shifting down each node’s neighbors are using at most 2 colors. This eliminates color 4. We then repeat this process to eliminate color 5: again shifting down, and then nodes with color 5 choosing a free color in \(\{1, 2, 3\}\). We again repeat to eliminate color 6.

6.3.2 \(\Delta + 1\)-coloring of bounded degree graphs

The deterministic symmetry breaking approach can be used to compute a \(\Delta + 1\)-coloring of any graph, not just rooted trees as follows. Consider a vertex \(v\), and let it have \(k\) (at most \(\Delta\)) neighbors: \(u_1, u_2, \ldots, u_k\). Initially, each node takes its own ID as its color; hence the size of initial color is \(O(\log n)\) bits. Then in each iteration, \(v\) applies the symmetry breaking rule with respect to each of its neighbors as follows and the final color is combined. Let \(<c(v)>\) be the bit representation of \(v\)’s color and, for \(1 \leq j \leq k\), let \(<c(u_j)>\) be the bit representation of the colors of its neighbors. Let \(b(u_j)\) be the index of the first (least significant) bit where \(<c(v)>\) differs from \(<c(u_j)>\), for \(1 \leq j \leq k\). Then
v sets its (new) color to be \(<b(u_1)>c_{b(u_1)}(u_1)\cdots<b(u_k)>c_{b(u_k)}(u_k)\>)

i.e., the concatenation of the \(<b(u_j)>c_{b(u_j)}(u_j)\) of all its neighbors. In the next step, the symmetry breaking rule is applied to each component individually, just as we did for rooted trees — we can imagine that instead of having just one parent for a node as in rooted trees, here there are \(\Delta\) parents and the reduction is done for each neighbor independently; the (final) color of a node is simply a concatenation of all the \((\Delta)\) reduced colors.

As in the case of Theorem 6.2, we can show that the above rule preserves legality while reducing the size of the color from \(\ell\) to at most \(\Delta([\log \ell] + 1)\) in one step. Thus by applying this reduction \(O(\log^* n)\) times, the the size of the color (i.e., the number of bits needed to represent the color) reduces to at most \(3\Delta\) (Exercise 6.9 asks you to show this). Hence the total number of colors in the graph is at most \(2^{3\Delta}\). We next discuss a simple technique to reduce the number of colors to \(\Delta + 1\) in \(O(2^{3\Delta})\) rounds.

We accomplish the reduction as follows which proceeds round by round, whereby, in each round, one color higher than \(\Delta + 1\) is eliminated, while still preserving a legal coloring. In the first round, we recolor all nodes with color (say) \(\Delta + 2\). This can be accomplished in one round, where only nodes with color \(\Delta + 2\) recolor. Each such node can choose a “free” color in the set \(\{1, 2, \Delta + 1\}\), since, the number of neighbors is at most \(\Delta\) and a free color (that is a color not used by any of its neighbors) is available. Since all nodes having color \(\Delta + 2\) are not adjacent to each other (because of the legality of the initial coloring), we are left with a legal coloring (there is no conflict when these nodes choose simultaneously). We then repeat the process in the next round to recolor all nodes having color \(\Delta + 3\), and so on. Hence in at most \(O(2^{3\Delta} - \Delta - 1)\) rounds we are done.

Thus we can state the following theorem and its corollary.

**Theorem 6.3.** Any graph with maximum degree \(\Delta\) can be colored in \(O(2^{3\Delta} + \log^* n)\) rounds using at most \(\Delta + 1\) colors.

**Corollary 6.1.** Any bounded degree graph with maximum degree \(\Delta\), where \(\Delta = O(1)\), can be in colored in \(O(\log^* n)\) rounds using at most \(\Delta + 1\) colors.

### 6.4 Dominating Set

The last problem that we will consider in this chapter is dominating set (defined in Section 6.1), another important local symmetry breaking problem. The goal is to find a dominating set that is of small size. Small dominating sets have many applications (mentioned in Section 6.1). Like MIS, they are a very useful
primitive used as a building block in other algorithms. For example, some of the efficient algorithms for minimum spanning tree (MST) use small dominating sets as a building block. If one requires a maximal dominating set (MDS), then as discussed in Section 6.1, it is enough to run a MIS algorithm, since a MIS is also a MDS. However, a MDS may not be small compared to the minimum dominating set, i.e., a dominating set that is smallest in size among all dominating sets. Note that computing a minimum-size dominating set is NP-hard [1] (the best known sequential algorithms take exponential time). However, one can find an $O(\log \Delta)$-approximate dominating set (whose size is no larger than the optimum by a factor of $O(\log \Delta)$, where $\Delta$ is the maximum degree) in polynomial time. In fact, this can be done by a simple greedy algorithm described as follows: starting from the initial graph, find the vertex of the highest degree and include it in the dominating set; remove this vertex from the graph and its incident edges and recurse on the residual graph. In this section, we will show how to distribute this algorithm which looks inherently sequential. In particular, we will present a distributed algorithm that outputs a dominating set whose size no more than a factor of $O(\log \Delta)$ times the optimal set (in other words, the approximation ratio is $O(\log \Delta)$) in $O(\log n \log \Delta)$ rounds (assuming that all nodes know both $n$ and $\Delta$; otherwise if $\Delta$ is not known, then this boils down to a $O(\log^2 n)$-round algorithm).

### 6.4.1 Set Cover Problem

Dominating set is a special case of set cover problem defined as follows: Given a ground set $X$ and a collection of non-empty subsets of $X$, the goal is to cover the ground set by using a minimum number of sets from the collection. In the case of the dominating set problem, the ground set is the set of vertices and the subsets are the closed neighborhoods of all the vertices and we need to choose as few vertices to cover all the vertices of the graph.

We will show that the following greedy algorithm gives an $O(\log \Delta)$ approximation, where $\Delta$ is the size of the largest set. The algorithm proceeds in iterations. In each iteration, it includes the set (which is chosen from the set collection) that covers the most uncovered elements. Note that the greedy algorithm for dominating set mentioned earlier, is simply this greedy algorithm tailored to the dominating set problem. This algorithm appears to be inherently sequential. We will see shortly that it can be distributed with only a small loss in performance.
6.4. DOMINATING SET

Performance of Greedy Algorithm

We first show that the greedy (sequential) algorithm gives a $O(\log \Delta)$ approximation to the optimal.

**Theorem 6.4.** The greedy algorithm gives a set cover of size that is within $O(\log \Delta)$ factor of the optimal.

We give two proofs for the above theorem. The first is conceptually simple and straightforward. The second proof uses a charging argument and is useful in designing a distributed algorithm.

**First Proof**

**Proof.** Let $r_i$ be the number of elements that are uncovered at the beginning of iteration $i$ by the greedy algorithm. This means that $r_0 = n$. Let $OPT$ be the optimum set cover and let its size be $|OPT|$. At the start of iteration $i$, since the optimum covers all the elements with $|OPT|$ number of sets, there is at least one set which has at least $r_i/|OPT|$ elements. Hence, greedy will cover at least $r_i/|OPT|$ elements in iteration $i$. Thus the number of elements that remain uncovered at the beginning of iteration $i+1$ is

$$r_{i+1} \leq r_i - r_i/|OPT| = r_i(1 - 1/|OPT|).$$

From the above recurrence, we have $r_i = r_0(1 - 1/|OPT|)^i = n(1 - 1/|OPT|)^i$.

Plugging $f = |OPT|\ln(n/|OPT|)$ (where $\ln$ is the natural logarithm), we have

$$r_f \leq n(1 - 1/|OPT|)^{|OPT|\ln(n/|OPT|)} \leq ne^{-\frac{1}{|OPT|\ln(n/|OPT|)}} = |OPT|.$$ 

Hence after picking $f$ sets, there are only $|OPT|$ elements left that are (still) uncovered; this can be covered by picking at most $|OPT|$ more sets. Hence the number of sets picked by greedy is at most $|OPT|\ln(n/|OPT|) + |OPT| = O(|OPT|\ln(n/|OPT|))$. Since $|OPT| \geq n/\Delta$ (Exercise 6.14), we have that the approximation ratio of greedy to be $O(\ln(n/|OPT|)) = O(\log \Delta)$. \qed

**Second Proof** To bound the performance of a set cover algorithm we do the following accounting(charging): when the algorithm picks a set, its price (which is 1) is distributed equally to all the new (i.e., previously uncovered) elements it covers, i.e., each new element covered gets a price of $1/k$, where $k$ is the number of new elements covered. Each element is assigned a price only once, at the time it is covered by the algorithm. Note that, greedy chooses a set $S$ at each iteration that realizes the minimum unit price. For any subset $A$ of the ground
set $X$, let $g(A) = \sum_{e \in A} p(e)$. By our accounting, $g(X)$, the sum of the unit prices, is the total cost incurred by greedy.

**Proof.** For any subset $S$ of the collection of the sets given, we will show that $g(S) \leq H_{|S|}$, where $H_{|S|} = 1 + 1/2 + 1/3 + \cdots + 1/|S|$ is the harmonic sum. Sort the elements of $S$ according to the iteration when they are covered by greedy, breaking ties arbitrarily. Let $e_1, \ldots, e_k$ be this numbering. When greedy covers $e_i$, $p(e_i) \leq 1/(k - i + 1)$. This is because, since $e_i$ belongs to $S$, greedy would have chosen to cover $e_i$ by choosing a set (in the collection) that has at least $k - i + 1$ uncovered elements, since $S$ has these many elements still uncovered (it could have chosen $S$ itself, but no other set which has lesser number of uncovered elements). Hence, $g(S) = \sum_{i=1}^{k} p(e_i) \leq \sum_{i=1}^{k} 1/(k - i + 1) = H_k = H_{|S|}$.

We have, for any two sets $A$ and $B$, $g(A \cup B) \leq g(A) + g(B)$. Denoting OPT as an optimal cover and GREEDY as the set output by the greedy algorithm, we have

$$|GREEDY| = g(X) = g(\cup_{S \in OPT} S) \leq \sum_{S \in OPT} g(S) \leq \sum_{S \in OPT} H_{|S|} \leq \max_{S} \{H_{|S|}\} \sum_{S \in OPT} 1 \leq \max_{S} \{H_{|S|}\} |OPT| = \max_{S} \{O(\log |S|)\} |OPT| = O(\log \Delta) |OPT|.$$

\[ \square \]

**Distributing Greedy**

Greedy appears inherently sequential. We add one set in each iteration — the one that covers the largest number of uncovered elements. Intuitively, it will be good to add many large-sized sets in parallel to speed up the algorithm; however, adding all sets simultaneously speeds up the algorithm, but destroys the approximation, since they do not follow the greedy principle of choosing the largest number of uncovered elements.

The main idea to distribute greedy is to come up with a modified accounting scheme (see the second proof of Theorem 6.4) that distributed costs among the elements in a manner similar to (sequential) greedy. For every set $S$ selected by greedy, its cost will be distributed among a subset $T \subseteq S$ of at least $|S|/4$ uncovered elements (instead of always distributing it equally among all uncovered elements in the set, as done by greedy). Elements of $T$ will be charged only once. It is easy to adapt the analysis of the greedy algorithm to this
setting to show that the above accounting also gives a $O(\log \Delta)$ approximation (the approximation will be worse by a factor of 4). This modified scheme can be more easily parallelized — the intuition is as follows. We consider nodes by bucketing them into $O(\log \Delta)$ categories, where all nodes in categories have approximately the same degree (within a factor of 2). For example, the first category has all nodes that have degree between $\Delta$ and $\Delta/2$. We process nodes in increasing order of categories (starting with the first category). Nodes that belong to the same category are processed in parallel. Even though all nodes in the same category have the approximately the same degree, they all cannot be included in parallel since it will violate the greedy rule. Instead, we include all nodes (in one category) if all of them cover a large enough fraction of their degree; this is what the modified accounting rule ensures. We show that one can process all nodes in a category in $O(\log n)$ rounds, giving an overall time bound of $O(\log n \log \Delta)$ rounds.

### 6.4.2 Distributed Algorithm for Dominating Set

We can think of the distributed algorithm for choosing the dominating set as an election. Nodes (voters) vote for one of the candidates, i.e., nodes in their respective closed neighborhoods. In other words, each node that is not dominated will choose one of its neighbors (including itself) as its dominating candidate. Note that, each node can be both a voter as well as a candidate (this is true in many real-world elections as well!!).

The algorithm is a sequence of $\log \Delta$ phases. In phase $i = 1, 2, \ldots, \log \Delta$, the maximum degree of the graph is at most $\Delta/2^{i-1}$. The candidates during phase $i$ are all those vertices whose degree is in the interval $(\Delta/2^i, \Delta/2^{i-1}]$. Initially all vertices are free. When a vertex becomes a dominator it does not participate in the algorithm any longer. The voters are free nodes. Candidates can be free or dominated vertices. Each phase consists of $O(\log n)$ elections, i.e., after $O(\log n)$ election iterations, one phase ends and the next begins. Each election iteration (described next) runs in $O(1)$ rounds. Hence totally we have $O(\log n \log \Delta)$ rounds. (If $\Delta$ is not known then the run time will be $O(\log^2 n)$ rounds.)

We now describe one election iteration in a phase. A candidate that is chosen by many voters enters the dominating set. The steps (which constitute one iteration of the election) are as follows. As long as some node is not dominated, it will participate in the election process. Without loss of generality, we assume that the graph does not have isolated nodes (these can be handled by simply including them).
1. Each node chooses a random rank in \([0, 1]\) (or \([0, n^3]\)).

2. Among all the candidate sets that contain it, each voter votes for that set which has the lowest rank.

3. A candidate is elected if it obtains at least 1/4 of the votes of its electorate (electorate is the set of nodes that can vote for it — they will be a subset of its closed neighborhood).

4. Elected candidates (nodes) enter the dominating set and they (and their incident edges) are deleted from the graph. (Nodes that have degree 0 are also deleted from the graph).

We note that the cost of the set can now be distributed equally among the elements that voted for it; thus the above steps implement the (modified) accounting scheme mentioned above. It is easy to see that all the above steps can be implemented in \(O(1)\) rounds (even in the CONGEST model).

**Analysis**

We analyze one iteration of the election in a (arbitrary) phase \(i\) and show that it finishes in \(O(\log n)\) rounds whp. For the sake of analysis, it is convenient to construct a bipartite graph: candidates on one side and its neighbors on the other. Note that the candidates during phase \(i\) are all those vertices whose degree is in the interval \((\Delta/2^i, \Delta/2^{i-1}]\); however, all non-dominated neighbors of candidates (no restriction on degree) participate during the phase. The neighbors of a candidate \(c\) form the electorate of \(c\). The neighbors of a voter \(v\) form the pool of \(v\).

We will show that the expected number of edges that are removed from the bipartite graph is a constant fraction in each iteration. This will imply that \(O(\log n)\) elections are enough to end a phase whp.

**Definition 6.4.1.** A voter is influential for a candidate \(c\) if at least 3/4 of the voters in \(c\)'s electorate have degree no greater than that of \(v\). Let \(d(v)\) denote the degree of \(v\) in the bipartite graph.

**Lemma 6.13.** For any two voters \(v\) and \(w\), \(d(v) \geq d(w)\), in \(c\)'s electorate,

\[
\Pr(w \text{ votes } c | v \text{ votes } c) \geq 1/2.
\]

**Proof.** Define the following:
6.4. DOMINATING SET

- Let $N_b$ denote the number of neighbors that $v$ and $w$ have in common.
- Let $N_v$ denote the number of neighbors of $v$ that are not neighbors of $w$.
- Let $N_w$ denote the number of neighbors of $w$ that are not neighbors of $v$.

Then
\[
\Pr(w \text{ votes } c | v \text{ votes } c) = \frac{\Pr(w \text{ votes } c \text{ and } v \text{ votes } c)}{\Pr(v \text{ votes } c)} = \frac{1/(N_v + N_b + N_w)}{1/(N_v + N_b)} \geq 1/2.
\]

\[\square\]

**Lemma 6.14.** Let $v$ be an influential voter for candidate $c$. Then
\[
\Pr(c \text{ is elected } | v \text{ votes } c) \geq 1/6.
\]

**Proof.** Let random variable $X$ denote the number of votes for $c$. Let $Y = C - X$, where $C$ is the size of the electorate of $c$; random variable $Y$ denotes the number of voters that did not vote for $c$.

\[
E[X|v \text{ votes } c] \geq \sum_{w : d(w) \leq d(v)} \Pr(w \text{ votes } c|v \text{ votes } c).
\]
\[
\geq \frac{13c}{24} = \frac{3c}{8}.
\]

The above follows because voter $v$ is influential for candidate $c$, and thus at least 3/4 of the voters in $c$'s electorate have degree no greater than that of $v$ and we use Lemma 6.13.

Hence,
\[
\Pr(c \text{ not elected } | v \text{ votes } c) = \Pr(X < c/4 | v \text{ votes } c)
\]
\[
= \Pr(Y \geq 3c/4 | v \text{ votes } c).
\]

Applying Markov’s inequality we have,
\[
\Pr(Y \geq 3c/4 | v \text{ votes } c) \leq \frac{E[Y | v \text{ votes } c]/(3c/4)}{4(c - E[X | v \text{ votes } c])/(3c)}
\]
\[
\leq 5/6.
\]
\[\square\]
Lemma 6.15. In a phase, let $m'$ denote the total number of edges in the bipartite graph at the beginning of the iteration (in this phase). Let $X$ denote the number of edges removed from the bipartite graph after one election. Then $E[X] \geq m'/24$.

Proof. An edge $(v, c)$ is good if $v$ is influential voter for candidate $c$. By the definition of an influential voter, at least $1/4$ edges are good.

We have

$$E[X] \geq \sum_{(v, c)} \Pr(c \text{ is elected and } v \text{ votes } c)d(v)$$

$$\geq \sum_{(v, c) \text{ is good}} \Pr(c \text{ is elected and } v \text{ votes } c)d(v)$$

$$= \sum_{(v, c) \text{ is good}} \Pr(v \text{ votes } c)\Pr(c \text{ is elected | } v \text{ votes } c)d(v)$$

Since $\Pr(v \text{ votes } c) = 1/d(v)$ and by applying Lemma 6.14 we have,

$$E[X] \geq m'/24.$$

The above lemma shows that, on average, a constant fraction of edges are deleted from the bipartite graph. Using techniques similar to those used in Section 6.1 (in particular, in the proof of Lemma 6.7), this leads to the following lemma.

Lemma 6.16. A phase terminates in $O(\log n)$ rounds whp.

Since there $O(\log \Delta)$ phases, we have the following time bound.

Theorem 6.5. The distributed algorithm for dominating set runs in $O(\log n \log \Delta)$ rounds with high probability.

6.5 Worked Exercises

Worked Exercise 6.1. Let $G = (V, E)$ be a graph where the maximum degree $\Delta$ is bounded by a constant. Give a MIS algorithm that runs in $O(\log^* n)$ rounds in $G$. 
6.5. WORKED EXERCISES

Solution: First run the coloring algorithm for a bounded degree graph (Section 6.3.2). The algorithm colors the graph $G$ by at most $\Delta + 1$ colors and finishes in $O(\log^* n)$ rounds (cf. Corollary 6.1).

Now we use the relation between independent set and node coloring. Notice that the set of nodes with the same color is an independent set. However, that set is not necessarily a MIS. Nonetheless, starting with a coloring, one can easily derive a MIS algorithm: Suppose the colors are numbered from $1, 2, \ldots, \Delta + 1$. In the first round all nodes of the first color join the MIS and notify their neighbors. Then, all nodes of the second color which do not have a neighbor that is already in the MIS join the MIS and inform their neighbors. This process is repeated for all colors. Hence, it takes at most $O(\Delta)$ rounds to compute a MIS from the $\Delta + 1$ coloring. Therefore the total time required is $O(\log^* n + \Delta)$ rounds. Since $\Delta$ is bounded by a constant, the required time bound follows.

Worked Exercise 6.2. Show that the MIS Algorithm 2 when run on a graph that is a path of length $n$ terminates in $O(\sqrt{\log n})$ rounds with high probability.

Solution: MIS Algorithm 2 terminates for a given node when it decides whether it is in the MIS or not. Define an undecided node as one which has not yet decided if it is in the MIS or not. We show that the maximum size of any path of consecutive undecided nodes drastically reduces after $2^{\sqrt{\log n}}$ rounds and then we bound the time taken for any node in any of these paths to decide.

Lemma 6.17. After $2^{\sqrt{\log n}}$ rounds, the maximum size of any path of consecutive undecided nodes is $2^{\sqrt{\log n}}$ w.h.p.

Proof. Consider a path of nodes of size $k + 1$. Let the ranks of the nodes from one end of the path to the other end be denoted by $n_1, n_2, \ldots, n_{k+1}$. Now, after one round of MIS Algorithm 2, $k$ nodes on this path will remain undecided iff either $n_1 > n_2 > \ldots > n_{k+1}$ or $n_1 < n_2 < \ldots < n_{k+1}$ (this is simple to prove). Thus the probability that a path of $k$ consecutive undecided nodes remains among this path is $\frac{2}{(k+1)!}$. The probability that the given path remains after $k$ rounds is $\left(\frac{2}{(k+1)!}\right)^k$ (because in each round, every node picks a new rank, so this condition must be met with the given probability in all $k$ rounds). There are $n-(k+1)$ possible such paths in a path of $n$ nodes (consider the starting position of the path). Thus by the union bound, the probability that there exists a path of $k$ consecutive undecided nodes remaining after $k$ rounds is upper bounded by $(n-k-1)\left(\frac{2}{(k+1)!}\right)^k \leq n\left(\frac{1}{k!}\right)^k$. If we set $k = 2^{\sqrt{\log n}}$, then the probability that there exists a path of $2^{\sqrt{\log n}}$ consecutive undecided nodes after $2^{\sqrt{\log n}}$ rounds
\[ \leq n \left( \frac{1}{(2\sqrt{\log n})!} \right)^{2\sqrt{\log n}} \]

\[ \leq n \left( \frac{1}{(2\sqrt{\log n}/2)(2\sqrt{\log n}/2)} \right)^{2\sqrt{\log n}} \]

\[ = \frac{1}{n^{\log \log n}} \]

\[ \leq \frac{1}{n^{1+c}} \text{ for a positive constant } c \text{ when } n \text{ is sufficiently large} \]

Therefore the probability that such a path of size \(2\sqrt{\log n}\) exists after \(2\sqrt{\log n}\) rounds is \(\frac{1}{n^{1+c}}\). The probability of maintaining a path of length \(> k\) for \(k\) rounds is smaller than this value. Thus the probability that there exists any path of size \(2\sqrt{\log n}\) or larger is upper bounded by \(n^{\frac{1}{n^{1+c}}} = \frac{1}{n^c}\), which is still very low. Thus the lemma is proved.

Now, consider any such path of consecutive undecided nodes. In every round, at least one node in the path will decide to join the MIS. Thus in at most \(2\sqrt{\log n}\) rounds, all nodes in every path of undecided nodes will decide. Thus, after \(O(\sqrt{\log n})\) rounds MIS Algorithm 2 terminates w.h.p.

### 6.6 Exercises

**Exercise 6.1.** The MIS algorithm 2 requires the knowledge of \(n\), the network size. Can you modify the algorithm so that it works without knowledge of \(n\).

**Exercise 6.2.** Compute the expected message complexity of the MIS algorithms 1 and 2. Also show a high probability bound on the message complexity.

**Exercise 6.3.** Give an \(O(n)\)-round deterministic algorithm for MIS in the synchronous CONGEST model.

**Exercise 6.4.** Consider the following algorithm for coloring a graph using \(\Delta + 1\) colors, where \(\Delta\) is the maximum degree of the graph. Let the colors be numbered from 1 to \(\Delta + 1\). Start with color 1 and an arbitrary node. Color that node with color 1 and an arbitrary node. Color that node with color 1, skip its neighbors, choose an arbitrary node from the remaining set of nodes and color that node with color 1 and so on. Continue till no more node can be colored with color 1 (note that the color 1 nodes form a MIS). Then choose color
2 and repeat the above process on nodes that are not yet colored and so on. Show that the above algorithm gives a valid $\Delta + 1$ coloring.

**Exercise 6.5.** Can we color every graph using $\Delta$ colors? Justify your answer.

**Exercise 6.6.** In the proof of Lemma 6.11 show that the probability that $u$ and $v$ choose the same color is at most $1/|L_u| = 1/(d_u + 1)$.

**Exercise 6.7.** Consider the $\Delta + 1$-coloring algorithm discussed above without the sleeping step, i.e., all vertices are awake in every step and pick a tentative color uniformly at random from its list of colors. Show that this algorithm also finds a $\Delta + 1$-coloring in $O(\log n)$ rounds with high probability.

**Exercise 6.8.** The input is a 6-node oriented tree. The root $r$ has two children $u$ and $v$. Node $u$ has one child $w$ and node $v$ has two children $x$ and $y$. The ID's of these nodes (in decimal) are as follows: $\text{ID}_r = 104$, $\text{ID}_u = 110$, $\text{ID}_v = 51$, $\text{ID}_w = 170$, $\text{ID}_x = 35$, and $\text{ID}_y = 15$. Show the execution of the deterministic 6-coloring algorithm on oriented trees on this input. Make sure to use the correct number of bits in the representation of colors in each round of the algorithm and also make sure to terminate the algorithm after the appropriate number of rounds. Your answer should be a sequence of appropriately labeled illustrations of the given oriented tree.

**Exercise 6.9.** In a graph with maximum degree $\Delta$, show that the deterministic symmetry breaking rule reduces the size of the color of node from $\ell$ to at most $\Delta(\lceil \log \ell \rceil + 1)$ and, furthermore, by applying the rule $O(\log^* n)$ rounds, the size of the color of every node reduces to at most $3\Delta$ bits.

**Exercise 6.10.** The input is a graph $G = (V, E)$ whose vertices have already been assigned a proper vertex coloring $c : V \rightarrow \{1, 2, 3, 4\}$. In other words, each node knows its color, denoted by the local variable $c(v)$. Furthermore, $c(v) \in \{1, 2, 3, 4\}$ for all $v \in V$ and $c(u) \neq c(v)$ for all $\{u, v\} \in E$. Design and present a distributed algorithm in the CONGEST model running in 4 rounds for computing an MIS of $G$. Use pseudocode to describe your algorithm.

**Exercise 6.11.** A matching of a graph $G = (V, E)$ is a subset of edges $M \subseteq E$ such that no two edges in $M$ share a common vertex. A matching $M$ is maximal if no more edges can be added to $M$ while still keeping $M$ as a matching. Give an $O(\log n)$-round distributed algorithm for finding a maximal matching.

**Exercise 6.12.** A distance-2 coloring of a graph $G = (V, E)$ is a vertex coloring $c : V \rightarrow \{\}$ such that no two vertices at distance at most two from each other have the same color. For any node $v$, let $N_2(v)$ (called the 2-neighborhood of $v$) denote
the set of vertices at distance at most two from $v$. Note that $N_2(v)$ contains $v$ and all its neighbors. Let $\Delta_2$ denote the size of the largest 2-neighborhood in the graph.

Describe a randomized algorithm, running in the LOCAL model in $O(\log n)$ rounds (in expectation), that produces a distance-2 coloring of $G$ using $\Delta_2$ colors. Write your algorithm using pseudocode executed by a node $v$. Make sure that your pseudocode is clear and well-commented.

**Hint:** Mimic the $(\Delta + 1)$-coloring algorithm presented here.

**Exercise 6.13.** Consider the following (non-distributed) algorithm on an input graph $G = (V, E)$.

```
while ($V \neq \emptyset$) do
    $S \leftarrow$ all nodes of degree at most 2
    for each edge $e \in E$ do
        if $e$ has exactly one endpoint in $S$ then
            orient $e$ from its endpoint in $S$ to its endpoint in $V \setminus S$
        if $e$ has both endpoints in $S$ then
            orient $e$ arbitrarily
    $V \leftarrow V \setminus S$
    $E \leftarrow$ edges in $G[V]$
```

(a) Prove that if $G$ is a tree, then this algorithm can be implemented (in the CONGEST model) to run in $O(\log n)$ rounds, yielding an orientation of edges such that every node has at most 2 out-neighbors.

(b) Show that there is a deterministic algorithm in the CONGEST model to compute a 3-coloring of an unoriented tree in $O(\log n)$ rounds.

**Exercise 6.14.** Show that in the set cover problem, $|OPT| \geq n / \Delta$.

**Exercise 6.15.** Adapt the second proof of Theorem 6.4 to show that the modified accounting scheme described in Section 5.4.1 gives a $O(\log \Delta)$ approximation.

**Exercise 6.16.** Time division multiple access (TDMA) is a method for different nodes in a distributed system to access a shared physical medium. For example, for nodes in a wireless network, their radio frequency channel is the shared medium that they all need to access to be able to communicate. TDMA is also used in other settings such as bus networks, where different nodes need to access a shared bus in order to communicate. In TDMA, time is partitioned into time slots and each node has its own time slot during which it uses the shared medium.
Consider the situation in which there are \( n \) identical (here “identical” means that nodes do not have IDs) nodes in a wireless network, each of which needs to send a message to a base station. The base station is within the transmission range of the wireless nodes, but is possible that not all pairs of wireless nodes are in each others’ transmission ranges. The wireless nodes have access to a single radio frequency channel and therefore if two or more of these nodes transmit their message at the same time, the base station hears a “collision,” but does not receive any of the transmitted messages. More precisely, a base station has the ability to distinguish among three situations: (i) it hears no transmission, (ii) it hears a “collision”, and (iii) it hears a message. The base station can also transmit, but it also uses the same radio frequency channel as the wireless nodes. Therefore, if the base station is transmitting at the same time as a wireless node, it will hear a collision.

Now notice that \( n \) nodes have to send messages to the base station and no two nodes can transmit at the same time. So at least \( n \) time slots are needed for all the messages to reach the base station. The problem is how should the wireless nodes and the base station coordinate transmissions so that it does not take too many time slots for all messages to reach the base station.

(a) Design a randomized algorithm that uses \( O(n) \) times slots in expectation to ensure that the messages from all \( n \) wireless nodes reach the base station. You can assume that nodes initially know \( n \).

(b) Prove that your algorithm runs in expected \( O(n) \) time slots.

(c) How many time slots will your algorithm take with high probability?
Let 1, 4, 5 be marked nodes according to probability $\frac{1}{2}d(v)\frac{d(1)}{d(5)}\frac{d(4)}{d(3)}\leq \frac{d(5)}{d(4)}\frac{d(3)}{d(2)}$. So, 4 unmarks itself.

2, 4 and 6 are also deleted, since 5 joins MIS.

$d(3) = 0$, So, status = yes

---

Figure 6.2: Illustrating MIS Algorithm 1
Figure 6.3: Illustration for proof of Lemma 6.1.
Chapter 7

Minimum Spanning Tree

In this chapter, we focus on one of the most central problems in distributed computing, the minimum spanning tree (MST) problem. The MST along with leader election are the most fundamental of the “global” distributed network problems. They are “global” because they both need at least $\Omega(D)$ rounds in a network of diameter $D$ (lower bounds are the focus of a later chapter), i.e., their computation needs the entire graph to be traversed.

The MST is an important and commonly occurring primitive in the design and operation of communication networks. Formally, the MST problem is, given a $n$-node connected graph with edge weights, the goal is to compute a spanning tree of minimum total weight, i.e., the total weight of the $n-1$ spanning tree edges. In practice, the weights can be used to model delay, congestion etc and hence an MST gives a spanning tree that minimizes total delay, congestion etc. One of the most common applications of MST is that can serve a backbone for efficient communication, e.g., it can be used naturally for broadcast. Any node that wishes to broadcast simply sends messages along the spanning tree. The advantage of this method over flooding is that redundant messages are avoided. The message complexity is $O(n)$ which is optimal. And since the spanning tree is one of minimum weight the total cost (assuming, weights model delay etc.) is minimized. A distributed MST algorithm can be also be used for leader election if a rooted tree (where parent-child relationships are known to each node) is constructed, since the root can serve as the leader. The MST algorithms described in this chapter naturally construct a rooted tree.

In this chapter, we present distributed algorithms for the MST problem. As usual, we assume the clean network model and the synchronous CONGEST model. (Recall that in the clean network model each node has only knowledge

\footnote{Note that all spanning trees of a connected $n$-node graph will have $n-1$ edges.}
only about itself and the weights incident on its edges; it does not know any
information about its neighboring nodes; CONGEST model means that only
$O(\log n)$-sized message is allowed to travel per round per edge.) Let’s discuss
the output of a distributed MST algorithm. At the end of the distributed MST
algorithm, each node will know which of its incident edges belong to the MST
and which do not, i.e., each node needs to know (only) the status of its incident
edges (and not other edges).

We make an assumption that simplifies our algorithms and analysis: we
assume that all edge weights in the graph are distinct. It is easy to show that
this implies that the MST is unique (Exercise 7.1). This assumption is without
loss of generality, because one can tag each edge weight (additionally) with the
IDs of the endpoints of the edge (which are unique as a pair)\footnote{Note that this tagging involves nodes knowing the IDs of their neighbors. This tagging can be
done during the course of an MST algorithm.}. This tagging can
be used to break ties between edges having the same weight.

As in centralized MST algorithms, distributed MST algorithms also rely on
two important properties of a MST: (1) \textit{cut property} and (2) \textit{cycle property}.

1. \textbf{Cut property}: A cut in a graph is a partition of the vertex set into two
disjoint sets. The cut property states that, given any cut in a graph, the
\textit{lightest} (minimum weight) edge crossing the cut belongs to the MST (due
to the assumption of unique edge weights, there is a unique lightest edge
crossing the cut.)

2. \textbf{Cycle property}: Consider any cycle in the graph. The heaviest (maximum
weight) edge in the cycle will not be in the MST.

\section{Gallager-Humblet-Spira (GHS) Algorithm}

The first distributed algorithm for the MST problem was given by Gallager,
Humblet, and Spira in 1983. This algorithm is known as GHS algorithm. We
next discuss the GHS algorithm.

We are given an undirected, connected, weighted graph $G = (V, E, w)$. Let
$n$ be the number of nodes and $m$ be the number of edges of $G$. Let $T$ be the
(unique) MST on $G$. A \textit{MST fragment} (or simply a fragment) $F$ of $T$ is defined
as a connected subgraph of $T$, that is, $F$ is a subtree of $T$. An \textit{outgoing edge}
of a MST fragment is an edge in $E$ where one adjacent node to the edge is in
the fragment and the other is not. The \textit{minimum-weight outgoing edge (MOE)}
of a fragment $F$ is the edge with \textit{minimum weight} among all outgoing edges
of $F$. As an immediate consequence of the cut property for MST, the MOE of a
7.1. GALLAGER-HUMBLET-SPIRA (GHS) ALGORITHM

fragment $F = (V_F, E_F)$ is an edge of the MST. The reason is as follows. Consider a cut $(V_F, V - V_F)$ of $G$. The MOE of $F$ is the minimum-weight edge in the cut $(V_F, V - V_F)$, and therefore the MOE is an edge of the MST. Thus adding the MOE of $F$ to $F$ along with the node at the other end of MOE yields another fragment of the MST.

The GHS algorithm operates in phases. In the first phase, the GHS algorithm starts with each individual node as a fragment by itself and continues till there is only one one fragment — the MST. That is, at the beginning, there are $|V|$ fragments, and at the end of the last phase, a single fragment which is the MST. All fragments find their MOE simultaneously in parallel.

In each phase, the algorithm maintains the following invariant: Each MST fragment has a leader and all nodes know their respective parents and children. The root of the tree will be the leader. Initially, each node (a singleton fragment) is a root node; subsequently each fragment will have one root (leader) node. Each fragment is identified by the identifier of its root — called the fragment ID — and each node in the fragment knows its fragment ID.

![Figure 7.1: GHS algorithm: Finding MOE of all fragments in parallel](image)

7.1.1 One phase of GHS

We describe one phase of the GHS. Each fragment’s operation is coordinated by the respective fragment’s root (leader). Each phase consists of two major operations: (1) Find MOE of all fragments and (2) Merging fragments via their MOEs.
Find MOE of all fragments

In a phase, all fragments find their MOE simultaneously in parallel. See Figure 7.1 for a illustration. To find the MOE of a fragment, the root in the fragment broadcasts a message (“find MOE”) to all nodes in the fragment using the edges in the fragment. Once a node receives “find MOE” message, it finds its minimum outgoing incident edge (i.e, the minimum weight outgoing edge among all the incident edges). To find the minimum weight outgoing incident edge, a node checks its neighbors in increasing order of weight. If the fragment ID of its neighbor is different from its own, then the edge is an outgoing edge. Note that since edges are checked in increasing order of weight, the first neighbor whose fragment ID is different from its own is the minimum outgoing incident edge. Also, note that the checking can be done (in increasing weight order) starting from the neighbor that was checked last in the previous phase. This is because, all edges that were checked earlier would belong to the same fragment and will continue to be in the same fragment till the end of the algorithm. Then, each node sends its minimum outgoing incident edge to the root by convergecasting the minimum; the root then finds the MOE, which is the minimum among all the edges convergecast. Note that the convergecast process uses the fragment (tree) edges only.

Merging fragments via their MOEs

In this phase, fragments are merged via their MOEs. All MOEs belong to the MST (by the cut property).
Once the leader finds the MOE, it sends broadcasts a “Merge” message to all its fragment nodes (the broadcast is sent along the tree edges); the message contains the MOE edge of the fragment. Hence a node when it receives the MOE edge, knows whether it is the same as its minimum outgoing incident edge or not. If a node has the MOE edge incident on it, then the node attempts to combine with its neighbor (which belongs to a different fragment). It sends a “Request to combine” message to its neighbor. If the neighbor has also selected the same edge as its MOE then the two neighboring nodes agree to combine through this edge; i.e., both neighboring nodes receive “Request to combine” message from each other. (Otherwise, if only one node receives this message, it ignores it; but the MOE edge is not ignored, the neighbor marks the edge over which it receives the message as the MOE edge of its neighboring fragment). The node with the higher identifier becomes the root of the combined fragment. The (combined) root broadcasts a "new-fragment" message through the fragment edges and the MOE edges chosen by all the fragments. Each node updates its parent, children, and fragment identifier (which will be ID of the new root).

To see that the above process correctly combines the fragments, we ascribe a direction to all the MOEs (towards the outgoing way). This creates a “directed tree” of fragments (think of fragments as “super-nodes”). See Figure 7.2 for an illustration. Note that since each fragment has only one outgoing edge, there can at most one pair of neighboring nodes (in this directed tree). One of these nodes in the pair will be the root of the combined fragment as described above.

### 7.1.2 Analysis of GHS algorithm

It is easy to argue that the total number of phases is \(O(\log n)\). This is because, in each phase, the total number of fragments is reduced by at least half: in the worst case, each MOE will be the MOE of both neighboring fragments and they combine into one.

We next argue that each phase takes \(O(n)\) time. Hence, overall time complexity is \(O(n \log n)\). This is because in each phase, both the major operations take \(O(n)\) time (these include broadcast and convergecast), since they happen along the MST edges. The diameter of the MST is at most \(O(n)\).

We next argue that each phase takes \(O(n)\) messages (for convergecast and broadcast) plus the messages needed to find MOE. The latter takes a total of \(O(m + n \log n)\) messages because in each phase a node checks its neighbor in increasing order of weight starting from the last checked node. Thus, except for the last checked node (which takes one message per phase) all other neighbors
are checked at most once. Hence total message complexity is
\[ \sum_{v \in V} 2d(v) + \sum_{i=1}^{\log n} \sum_{v \in V} 1 = O(m + n \log n). \]

Thus we can show the following theorem.

**Theorem 7.1.** The GHS algorithm described above correctly a distributed MST in \( O(n \log n) \) rounds and uses \( O(m + n \log n) \) messages.

### 7.2 The Pipeline Algorithm

In this section, we present an algorithm that is slightly better than the GHS algorithm, i.e., it runs in \( O(n) \) rounds. Note that this is existentially optimal, since we will show (later) that the diameter is a lower bound for MST (even for randomized algorithms) and hence there exists graphs of diameter \( \Theta(n) \), where any MST algorithm will require \( \Omega(n) \) rounds.

The Pipeline algorithm is essentially an upcast algorithm, where we build a BFS tree over the graph and each node upcasts edges to the root of the BFS tree; the root ends up having (enough) global knowledge of the network topology and locally computes the MST and downcasts the MST edges to all nodes in the network. Of course, a naive upcast is for each node to send all its incident edges and this upcast can take \( \Theta(m) \) rounds, since there are as many edges. The main idea of the Pipeline MST algorithm is to filter the number of edges broadcast so that the running time is reduced to \( O(n) \) rounds. However, as we will see the message complexity of the Pipeline algorithm can be as much as \( \Theta(n^2) \).

The pipeline algorithm uses the cycle property of MST to filter edges at intermediate nodes. Each node \( v \), except the root \( r \), maintains two lists of edges, \( Q \) and \( U \). Initially, \( Q \) contains only the edges adjacent to \( v \), and \( U \) is empty. At each round, \( v \) sends the minimum-weight edge in \( Q \) that does not create a cycle with the edges in \( U \) to its parent and moves this edge from \( Q \) to \( U \). If \( Q \) is empty, \( v \) sends a terminate message to its parent. The parent after receiving an edge from a child, adds the edge in its \( Q \) list. A leaf node starts sending edges upwards at round 0. An intermediate node starts sending at the first round after it has received at least one message from each of its children. Algorithm 7.3.2 gives the pseudocode.

#### 7.2.1 Analysis

Before we go to the formal proof, we give the high-level idea behind the correctness and running time.
Algorithm 13 Pipeline algorithm

1: Build a breadth-first tree $B$ in $G$. Let $r$ be the root of $B$.
2: Each node $v$, except the root $r$, maintains two lists of edges, $Q(v)$ (candidate set) and $U(v)$.
3: A leaf node starts sending edges upwards at round 0. An intermediate node starts sending at the first round after it has received at least one message from each of its children.
4: Initially, $Q(v)$ contains only the edges adjacent to $v$, and $U(v)$ is empty. At each round, $v$ sends the minimum-weight edge in $Q(v)$ that does not create a cycle with the edges in $U(v)$ to its parent and moves this edge from $Q(v)$ to $U(v)$. Any edge that creates a cycle with edges in $U(v)$ is deleted from $Q(v)$. If $Q(v)$ is empty, $v$ sends a terminate message to its parent. The parent after receiving an edge from a child, adds the edge in its $Q(v)$ list.
5: The root $r$ computes the MST locally from among the edges it hears from its children. The solution is then broadcast over tree $B$ to all nodes.

Correctness. The algorithm’s correctness follows from the cycle property. Using the cycle property, since only non-MST edges are filtered (note that an edge at node $v$ is not sent upward if it closes a cycle with edges in $U$, i.e., the already sent edges — since edges are sent in non-increasing order, the filtered edges are the heaviest edge in a cycle) it can be shown that the root receives all the MST edges (plus may be additional edges) required to compute the MST correctly.

Running time. It is easy to show that the edges reported by each node to its parent in the tree are sent in nondecreasing weight order, and each node sends at most $n - 1$ edges upward to its parent. This is because if more than $n - 1$ edges are sent through a node, then at least one edge will form a cycle with the edges sent previously and will be filtered by the cycle property. To build the BFS tree, it takes $O(D)$ time. Since the depth of the BFS tree is $D$ and each node sends at most $n - 1$ edges upward, the pipeline algorithm takes $O(D + n) = O(n)$ time. The analysis shows that there is not too much delay (more than $n$) overall before the root gets all the edges that it needs to compute the MST.

We now formally argue the correctness and running time. We make two simple observations:

1. The edges reported by each intermediate vertex to its parent in the tree are cycle-free.
2. Every vertex starts sending messages upwards at round $L(v) = \text{Depth}(T(v))$.

We define an active child which we use in the analysis.
**Definition 7.2.1.** Consider an intermediate vertex \( v \) at height \( h \) that has still not terminated its participation in the algorithm, at round \( t \), for some \( t \geq h \). A child is active if it has not terminated yet (i.e., it sent an element to \( v \) in round \( t - 1 \)).

The next lemma is the key lemma.

**Lemma 7.1.** (a) For each child \( u \) of \( v \) that is still active at round \( t \), \( Q(v) \) at the beginning of round \( t \) contains at least one edge.

(b) If \( v \) sends to its parent an edge of weight \( w_0 \) at round \( t \), then all of the elements \( v \) was informed at round \( t - 1 \) by its active children were of weight \( w_0 \) or larger.

(c) If \( v \) sends to its parent an element of weight \( w_0 \) at round \( t \), then any later element it will learn is of weight \( w_0 \) or larger.

(d) Any non-root node \( v \) sends elements in nondecreasing weight order to its parent; it sends the elements in a continuous fashion till it terminates.

**Proof.** The proof is by induction on the height of the tree. The base case trivially holds for leaves.

We consider the induction step. Consider an intermediate vertex \( v \) at height \( h \) and assume that the claims hold for each of its children.

**Claim (a) :** Let \( A_v \) be the set of \( k \) elements sent by \( v \) to its parent during the first \( t - h \) rounds (i.e., \( h \ldots, t - 1 \)). Note that the set of edges in \( A_v \) are cycle-free, by Line 4 of the Pipeline algorithm. Consider an active child \( u \) of \( v \). Let \( A_u \) be the set of elements sent by \( u \) to \( v \) up to round \( t - 1 \). Similarly, the set \( A_u \) is cycle-free. \( u \) has transmitted continuously to \( v \), since round \( L(u) \leq h - 1 \). Hence, \( |A_u| \geq k + 1 \). Thus there exists some edge \( f \in A_u - A_v \) such that \( A_v \cup \{f\} \) is cycle-free (see Exercise 7.1). This element belongs to \( Q(v) \).

**Claim (b) :** Consider any active child \( u \) of \( v \). Let \( f \) be the element sent by \( u \) on round \( t - 1 \). Let \( f' \) be some element sent by \( u \) at some round \( t' \leq t - 1 \) and is still in \( Q(v) \) at round \( t \). Then: \( w(f) \geq w(f') \geq w_0 \).

**Claim (c) :** This follows trivially from Claim (b).

**Claim (d) :** This claim follows from Claim (c) and the element selection rule, i.e., selecting the minimum weight edge in \( Q(v) \). Claim (a) shows \( v \) that the elements are sent continuously.
7.3. GARAY-KUTTEN-PELEG (GKP) ALGORITHM: AN $O(D + \sqrt{n} \log^* n)$-ROUND ALGORITHM

Lemma 7.2 (Termination lemma). After a vertex $v$ has terminated its participation in the algorithm, it will learn of no more reportable edges.

Proof. The proof is by induction. The base case is true for leaves. The induction step follows from Claim (a) of Lemma 7.1 as follows. Consider an intermediate node $v$. By Claim (a), it is not possible to have $v$ terminate at round $t$ (i.e., no more edges in $Q(v)$) unless all of its children have terminated by time $t$.

Running time We are now ready the finish the time analysis. The root starts getting messages at time $\text{Depth}(T)$. The root receives at most $n - 1$ elements from each of its children. By Lemma 7.1 these elements are sent in a fully pipelined fashion. The time to know all the edges is $O(n - 1 + \text{Depth}(T)) = O(n)$. The root can broadcast the entire set of MST edges in an additional $O(\text{Depth}(T) + n - 1) = O(n)$ time.

Message analysis We analyze the message complexity of the algorithm. In the worst case, each node can send $\Theta(n)$ edges upward, and hence the overall message complexity is $O(n^2)$.

Hence we can state the following theorem.

Theorem 7.2. The Pipeline algorithm correctly computes the MST in $O(n)$ time and using $O(n^2)$ messages.

7.3 Garay-Kutten-Peleg (GKP) Algorithm: An $O(D + \sqrt{n} \log^* n)$-round algorithm

We present a distributed MST algorithm runs in $O(D + \sqrt{n} \log^* n)$ time, where $D$ is the diameter of the graph $G$. Note that in graphs with diameter smaller than $\sqrt{n}$, this algorithm can be much faster than the Pipeline algorithm. We will show later that diameter $D$ is a fundamental lower bound for global problems such as leader election and MST in a “universal sense”, i.e., for every $D$ there exists graphs where $\Omega(D)$ is a lower bound for such problems. Given that, $D$ is an inherent lower bound parameter, this algorithm’s running time depends on $D$ in an explicit manner. This touches on an interesting aspect of distributed algorithms: identifying graph parameters (such as $D$) that can better capture the complexity of distributed network computations. For many networks, their diameter $D$ is significantly smaller than the number of vertices $n$, and therefore it is desirable to design protocols whose running time is bounded in terms of $D$ rather than in terms of $n$. 
We first give a high-level overview of the GKP algorithm; the details are in the following sections. The GKP algorithm consists of two parts: it combines the GHS algorithm and the Pipeline algorithm in a judicious way.

**First part: Controlled-GHS** The first part, called the controlled-GHS algorithm is similar to the GHS algorithm, with the crucial property of ensuring that the diameter of fragments do not become too big. (Note that in the GHS algorithm, even after the first phase, there can be fragments with diameter that are as large as $n$). The controlled-GHS begins with each node as a singleton fragment. In every phase, as in the GHS algorithm, each fragment finds its MOE. However, not all fragments are merged along their MOE edges. Only a subset of MOE edges are selected for merging. A crucial property of the merging is the following: in every phase, the number of fragments is reduced by at least a factor of two, while the diameter is not increased by more than a constant factor (the constant is important, as mentioned in Section 7.3.1). The controlled-GHS algorithm continues for about $\log^*(p n)$ phases. At the end of the first part, the following property is guaranteed: the diameter of each fragment is at most $O(\sqrt{n})$ and there are at most $p n$ fragments. The first part of the algorithm takes $O(p n \log^* n)$ time.

**Second part: Pipeline algorithm** The second part of the algorithm uses the Pipeline algorithm to find the remaining $\sqrt{n} - 1$ MST edges (since there are only $\sqrt{n}$ fragments left). As in the Pipeline algorithm, a breadth-first tree $B$ is built on $G$. Let $r(B)$ be the root of $B$. Using the edges in $B$, $r(B)$ collects weights of the interfragment edges, computes the minimum spanning tree, $T'$, of the fragments by considering each fragment as a super node. It then broadcasts the edges in $T'$ to the other nodes using the breadth-first tree $B$. Since the depth of $B$ is $D$ and each node sends at most $\sqrt{n}$ edges upward, the Pipeline algorithm takes $O(D + \sqrt{n})$ time; this analysis is exactly similar to the Pipeline algorithm (with $\sqrt{n}$ replacing $n$). Thus the overall time complexity of GKP algorithm is $O(D + \sqrt{n} \log^* n)$. We will show that the overall message complexity of this algorithm is $O(m + n^{1.5})$.

### 7.3.1 First part: Controlled-GHS algorithm

The pseudocode is given in Algorithm 14. Controlled-GHS is a modified variant of the original GHS algorithm, with the purpose of producing a balanced outcome in terms of number of resulting fragments and their diameter, whereas the original GHS algorithm allows an uncontrolled growth of fragments which can lead to
some fragments having a large diameter (as much as $\Theta(n)$). This is achieved by computing, in each phase, a maximal matching on the fragment forest and merging fragments accordingly.

We explain the merging procedure of the controlled-GHS algorithm in more detail. Initially, as in the GHS algorithm, there are $n$ singleton fragments. In each phase, each fragment finds its minimum outgoing (MOE) edge (i.e., the lightest edge connecting itself with the rest of the graph) as in the GHS algorithm (Section 7.1.1). However, unlike the GHS algorithm, in each phase, the merging is done in a more "balanced" fashion in the controlled-GHS algorithm, i.e., at the end of phase $i$, the diameter of every fragment is bounded by $O(2^i)$. The number of fragments at the end of phase $i$ is also bounded by $2^i$.

Let $F_i$ be the set of fragments at the beginning of phase $i$. (In the first phase, $F_1$ consists of $n$ singleton fragments.) Only fragments that have diameter at most $2^i$ will find MOE edges (unlike all fragments as in the GHS algorithm); the intuition is that this keeps the diameter of the merged fragments under control. Each fragment $F \in F_i$ of diameter at most $2^i$ determines the MOE of $F$ (finding MOE follows the same procedure as explained in the GHS algorithm — Section 7.1.1) and adds it to a candidate set $M_i$ of MOE edges. Consider the fragment graph $H_i = (F_i, M_i)$ defined as follows. The fragment graph consists of vertices $\{F_1, \ldots, F_k\}$, where each $F_j$ ($1 \leq j \leq k$) is a fragment at the start of phase $i$ of the algorithm. The edges of $H_i$ are obtained by contracting the vertices of each fragment $F_j \in F_i$ to a single vertex in $H_i$ and removing all resulting self-loops of $H_i$, leaving only the MOE edges in set $M_i$. Note that the fragment graph is a tree, in fact a rooted tree: direct each MOE edge of a fragment as an outgoing edge. We further note that the fragment is not explicitly constructed in the algorithm; it is just a construct to explain it.

As mentioned earlier, not all MOE edges are used for merging (as in the GHS algorithm); only a subset of them are used as described below. The algorithm first finds a maximal matching in the fragment graph $H_i$ (the intuition for using a maximal matching is mentioned below). Since the fragment graph is a rooted tree, this can be accomplished by running the deterministic symmetry breaking algorithm of Section 6.3 which runs in $O(\log^* n)$ rounds on the fragment graph (Exercise 7.3 asks you to show how this can be adapted to find a maximal matching instead of a coloring). However, note that since $H_i$ is obtained by contracting the fragment, the symmetry breaking algorithm has to be simulated by the leader of each fragment. Hence simulating one round on the fragment graph will take $O(\sqrt{n})$ rounds, since the diameter of each fragment is bounded by $O(\sqrt{n})$ (as shown below). Hence in the (original) graph, the algorithm takes $O(\sqrt{n} \log^* n)$ rounds.

The set of MOE edges chosen in the maximal matching are used for merging.
Let’s call this set $M'_i$ (Line 4 of Algorithm 14). The intuition for using only these matched edges is that there are no long chains. Indeed, only two fragments are merged by merging along the matched edges and hence the diameter increases only by a factor of 2. This is what we want, but we also want the number of fragments to go down by at least a factor of 2. If we use only the matched edges to join fragments, this may not happen. For example, let the fragment graph be a star graph. In this case, there is only one matched edge and only two fragments are merged. To avoid this kind of situation, the algorithm merges more edges: If a fragment of diameter at most $2^i$ has no incident matching edge (i.e., it has no matching edge touching it) it adds its MOE edge to the set of chosen edges $M'_i$ (Line 5 of Algorithm 14). Merging now takes place along the matching edges plus these additional added edges. Note that this increases the diameter of the merged fragment, since more fragments are merged; however, since only fragments that have no incident matching edges are merged, the increase in diameter is not too much. Thus we need two properties to hold: (1) the diameter of the merged fragment does not increase by too much; and (2) the number of fragments remaining at phase $i$ is at most $n/2^i$. We show these in the following lemmas.

**Lemma 7.3.** At the beginning of phase $i$, each fragment has diameter $O(2^i)$. In particular, at the end of the controlled-GHS algorithm each fragment has diameter $O(\sqrt{n})$.

**Proof.** We show that at the beginning of phase $j$, the diameter of each fragment is at most $5 \cdot 2^j$. The proof is by induction on $j$. The base case, i.e., at the beginning of phase 0, the statement is trivially true, since $5 \cdot 2^0 = 5 \cdot 2^0 = 4$ which is bigger than 0, the diameter of a singleton fragment.

By induction hypothesis, the diameter of each fragment at the beginning of phase $i$ is at most $5 \cdot 2^i$. We show that at the end of phase $i$ (i.e., at the beginning of phase $i+1$), the diameter of each fragment is at most $5 \cdot 2^{i+1}$.

In phase $i$, we compute an upper bound on the diameter of any resulting fragment. Note that a fragment consists of joining two fragments via a matching edge in the fragment graph. Note that at least one of these two has diameter at most $2^i$ since only fragments with diameter at most $2^i$ find MOE edges; the MOE edge may lead to a fragment with larger diameter, i.e., at most $5 \cdot 2^i$. In addition to joining two fragments via a matching edge, other fragments (with diameter at most $2^i$) can be joined to either fragments of the matching edge (see Figure 7.3). Hence the resulting diameter of the fragment at the end of phase $i$ is at most $5 \cdot 2^i + 3 \cdot 2^i + 3$, since the diameter of the combined fragment is determined by at most 4 fragments, out of which at most one has diameter $5 \cdot 2^i$ and the other three have diameter $2^i$ and these are joined by 3 MOE edges.
7.3. GARAY-KUTTEN-PELEG (GKP) ALGORITHM: AN O(D+\sqrt{N } \log^* N)-ROUND ALGORITHM

(which contributes to the constant 3). Hence the diameter at the end of phase \(i\) is at most
\[5 \cdot 2^i + 3 \cdot 2^i + 3 \leq 8 \cdot 2^i + 3 \leq 5 \cdot 2^{i+1},\] for \(i \geq 1\).

Since the controlled-GHS algorithm runs for \([\log \sqrt{\pi}]\) phases, the diameter of each fragment at the end of the algorithm is at most \(O(2^{[\log \sqrt{\pi}]}) = O(\sqrt{\pi})\).  

**Lemma 7.4.** At the beginning of phase \(i\), each fragment has size at least \(2^i\).

**Proof.** We prove the above statement via induction on \(i\). The base case \((i = 0)\) is trivially true, since at the beginning of phase 0, each fragment is of size \(2^0 = 1\).

Assume the statement is true for phase \(j\). We show that it holds for phase \(j+1\) as well. Let \(i = \lceil \log \pi \rceil\) and \(n = |V|\).

Consider phase \(j\) and assume \(i = \lceil \log \pi \rceil\).

Let \(\sqrt{\pi} < 2^j\), then \(i < j\) and \(\lceil \log \pi \rceil < \lceil j \rceil\), i.e., \(\pi < 2^{\pi} \leq 2^j\).

Thus, \(\pi \geq 2^{\pi}\) and at the beginning of phase \(j\), each fragment has size at least \(2^i\). Therefore, at the beginning of phase \(j+1\), each fragment has size at least \(2^j\).

**Corollary 7.1.** The number of fragments remaining at the beginning of phase \(i\) is at most \(\frac{n}{2^i}\). In particular, at the end of the controlled-GHS algorithm the number of fragments remaining is at most \(\sqrt{\pi}\).

**Proof.** From Lemma 7.4, at the beginning of phase \(i\), each fragment has size at least \(2^i\). Since fragments are disjoint, this implies that the number of fragments at the beginning of phase \(i\), is at most \(\frac{n}{2^i}\). Thus, after \([\log \sqrt{\pi}]\) phases, the number of fragments is at most \(\frac{n}{2^{[\log \sqrt{\pi}]}} \leq \sqrt{\pi}\).  

Thus we are ready the show the final lemma that shows the correctness and complexity of controlled-GHS algorithm.

**Lemma 7.5.** Algorithm \([14]\) outputs at most \(\sqrt{n}\) MST fragments each of diameter \(O(\sqrt{\pi})\) in \(O(\sqrt{\pi} \log^* n)\) rounds and incurs \(O(m + n \log^* n \log n)\) messages.

**Proof.** The correctness of the algorithm is established in Lemma 7.3 and Corollary 7.1.

We next show the complexity bounds. Consider one of the \(O(\log n)\) iterations of the algorithm. Similar to the analysis of the GHS algorithm (Section 7.1.2), the time complexity for finding the MOEs is \(O(m + n \log n)\). The time complexity for finding the MOEs in phase \(i\) is \(O(2^i)\), since the diameter of all
Algorithm 14 Controlled-GHS algorithm: Outputs at most $\sqrt{n}$ MST fragments each of diameter $O(\sqrt{n})$.

1: $\mathcal{F}_1 = (V, \emptyset)$ // initial set consisting of $n$ (singleton) fragments
2: for $i = 0, \ldots, \lceil \log \sqrt{n} \rceil$ do
3: Each fragment $F \in \mathcal{F}_i$ of diameter at most $2^i$ determines the MOE of $F$ and adds it to the candidate set $M_i$.
4: Find a maximal matching $M'_i \subseteq M_i$ in the fragment graph $H_i = (\mathcal{F}_i, M_i)$.
5: If $F \in \mathcal{F}_i$ of diameter at most $2^i$ has no incident edge in $M'_i$, it adds its MOE edge into $M'_i$.
6: $\mathcal{F}_{i+1}$ is obtained by merging all the fragments along the edges selected in $M'_i$.

Then a matching is built using the $O(\log^* n)$-deterministic symmetry-breaking algorithm. The symmetry breaking algorithm is simulated by the leaders of neighboring fragments by communicating with each other; since the diameter of each fragment is bounded by $O(2^i)$, the time needed to simulate one round of the symmetry breaking algorithm in phase $i$ is $O(2^i)$ rounds. Since only MST edges are used in communication, the total number of messages needed is $O(n)$ per round of simulation. Since there are $O(\log^* n)$ iterations, the total time and message complexity for building the maximal matching is $O(\sum_{i=0}^{\lceil \log \sqrt{n} \rceil} 2^i \log^* n) = O(\sqrt{n} \log^* n)$ and $O(n \log^* n)$ respectively. Afterwards, adding selected edges into $M'_i$ (Line 5 of the Controlled-GHS algorithm) can be done with additional $O(n)$ message complexity and $O(2^i)$ time complexity in phase $i$. Since there are $\lceil \log \sqrt{n} \rceil = O(\log n)$ phases in the controlled-GHS algorithm, the overall message complexity of the algorithm is $O(m + n \log^* n \log n)$. The overall time complexity is $O(\sqrt{n} \log^* n)$. \hfill \Box

7.3.2 Second Part: Combining the Remaining $\sqrt{n}$ fragments

The second part of the GKP algorithm combines the remaining at most $\sqrt{n}$ fragments using the Pipeline algorithm (Section 7.2) as follows. Recall that in the Pipeline algorithm (Algorithm ) we build a BFS tree first. (If a root or leader is known, then BFS construction takes $O(D)$ rounds and $O(m)$ messages — Section 3.7 Exercise 7.4 deals with the situation if there is no leader). In the algorithm described in Section 7.2, leaves of the BFS tree send their incident edges to their respective parents in increasing order of weight. Intermediate nodes forward the edges up the tree in increasing order of weight; the edges forwarded are cycle-free. In the present setting, since only at most $\sqrt{n}$ fragments remain, at most $\sqrt{n} - 1$ MST edges need to be discovered (unlike the case of
the original Pipeline algorithm where all the $n - 1$ MST edges needed to be discovered). Furthermore, not all incident edges need to be flooded, only inter-fragment edges need to be forwarded. Each node can identify its inter-fragment incident edges by checking the fragment ID of its neighbors. Only these edges are forwarded in increasing order. The filtering rule (via the cycle property) ensures that no node forwards more than $\sqrt{n} - 1$ edges, since any more will close a cycle. Hence, the message complexity is $O(n^{1.5})$. The Pipeline algorithm guarantees that the edges are sent continuously by each node in a pipelined fashion, hence the time complexity is $O(D + \sqrt{n})$, since the depth of the tree is $O(D)$ and $O(\sqrt{n})$ edges are sent in a pipelined fashion. The correctness follows from the correctness proof of the Pipeline algorithm (Lemma 7.1).

7.3.3 The Overall Algorithm

The overall algorithms consists of running the controlled-GHS first and then switching to the Pipeline algorithm after $O(\sqrt{n} \log^* n)$ rounds. Combining the two parts, we can show the following theorem.

Theorem 7.3. The GKP algorithm correctly computes a MST in $O(D + \sqrt{n} \log^* n)$ rounds and takes $O(m + n^{1.5})$ messages.

7.4 Worked Exercises

Worked Exercise 7.1. In a connected undirected graph with distinct edge weights, prove that there is a unique MST.

Solution: Suppose, to the contrary, that there are two different MSTs, $T$ and $T'$. Let $e$ be the minimum-weight edge that is in $T$ but not in $T'$. The graph $\{e\} \cup T'$ must contain a cycle, and at least one edge in this cycle, say $e'$, is not in $T$, as $T$ contains no cycles. Since the edge weights are all distinct and $e'$ is in one but not both of the trees, weight of $e$ is strictly less than the weight of $e'$. Thus $\{e\} \cup T' - \{e'\}$ is a spanning tree of smaller weight than $T'$; this is a contradiction.

Worked Exercise 7.2. Given a $n$-node complete network $G$ with edge weights, give a distributed algorithm to compute the MST of $G$ in $O(\log n)$ rounds. (Hint: Try to efficiently simulate the GHS algorithm.)

Solution: We implement the GHS algorithm which takes $O(\log n)$ phases. We will show how to implement one phase in $O(1)$ rounds, hence overall time is $O(\log n)$ rounds, as desired.
Before the start of the first phase, we do a pre-processing step: elect a leader for the entire graph — since we have a complete graph, this can be done in $O(1)$ rounds using $O(n^2)$ messages — each node broadcasts its ID to all other nodes and the minimum ID learnt is the leader.

Each phase consists of two parts: (1) Finding MOE and (2) Merging fragments. Note that each fragment is identified by all nodes in that fragment having the same fragment ID. Finding MOE and merging can be done as follows. Each node finds its minimum weight incident edge (which can be done by contacting all its neighbors to learn their fragment IDs) and sends that edge to the leader (along with its fragment ID and its own ID; note that the edge will be of the form $(f, g)$, where $f$ and $g$ are fragment IDs). The leader then aggregates the MOE edges (fragment by fragment) and finds the MOE edge per fragment (this can be done since the leader gets to know all the minimum incident edges from all nodes and their respective fragment IDs). Since the leader knows all the MOEs, it can merge locally, i.e., it can find out which fragments merge with which, and send the merged fragment IDs back to the respective nodes. The nodes rename their fragment IDs (if they got merged). This takes $O(1)$ rounds.

### 7.5 Exercises

**Exercise 7.1.** Given a weighted graph $G = (V, E, w)$, let $\mathcal{S}$ be a collection of cycle-free subsets of $E$ (i.e., a subset of edges which don’t contain a cycle) closed under inclusion (i.e., of $A \in \mathcal{S}$ and $B \subseteq A$ then also $B \in \mathcal{S}$). Note that the MST is the maximal cycle-free subset (i.e., a spanning tree) of largest weight.

Show that $\mathcal{S}$ satisfies the following important replacement property: If $A, B \in \mathcal{S}$ and $|B| = |A| + 1$, then there exists some element (edge) $f \in B - A$ such that $A \cup \{f\} \in \mathcal{S}$. In other words, there exists an edge $f$ belonging to $B$ (but not belonging to $A$) such that $f$ and the edges of $A$ are cycle-free. (Hint: Use the fact that if there are $\ell$ cycle-free edges in a graph with $n$ nodes, then the number of connected components is exactly $n - \ell$.)

**Exercise 7.2.** Give an input graph $G$ with $n$ nodes and $m = O(n)$ edges for which the GHS algorithm is tight in the number of messages used, i.e., it takes $O(n \log n)$ messages. Show your bound.

**Exercise 7.3.** Given a rooted tree, show how to find a maximal matching in $O(\log^* n)$ rounds.

**Exercise 7.4.** Show how to deterministically elect a leader at the beginning of the second part of the GKP algorithm. Note that the time and message complexity
of leader election should not exceed the overall time and complexity of the GKP algorithm. (Once a leader is elected, a BFS tree can be built which is used in the Pipeline phase).

Exercise 7.5. Show that in the GKP algorithm, the Pipeline phase can be implemented in $O(D + \sqrt{n})$ rounds using $O(m + n^{1.5})$ messages.

Exercise 7.6. Consider the following spanning tree verification problem: Given a graph $G = (V, E)$ with $n$ nodes and $m$ edges, and a subgraph $H$ of $G$, the goal is to verify whether $H$ forms a spanning tree of $G$. The goal is design a distributed algorithm that runs on $G$ (as usual, each node has only local knowledge: which of its incident edges belong to $H$ or not) that outputs “yes” (i.e., all nodes will be in state “yes” at the end) if $H$ is a spanning tree of $G$ and “no” otherwise (i.e., all nodes will be in state “no” at the end). Design a $O(D + \sqrt{n} \log^* n)$ algorithm for the problem, where $D$ is the diameter of $G$ (and not diameter of $H$). Show that your algorithm works. (Hint: Use GKP algorithm.)
Figure 7.3: Controlled-GHS algorithm: Merging fragments. The figure shows a fragment graph consisting of 8 fragments — a, b, c, d, e, f, g, h. In the fragment graph each fragment is represented as a node and the edges between fragments are the MOE edges — the outgoing edges are directed away from the respective fragments. The red edges are the edges of a maximal matching. The fragments are merged via the matched edges plus the added green edges; a green edge is the MOE edge of a fragment which has no incident matching edge — in the above example, these are fragments a and c. The figure shows the merged fragments after merging via the red and green edges.
Chapter 8

MapReduce Algorithms

8.1 Overview of MapReduce

MapReduce is a programming paradigm and an associated implementation suitable for processing large data sets on a cluster of commodity machines. This paradigm, and in particular its open source version Hadoop, has emerged as a de facto standard and is the tool of choice for large-scale data analysis at many companies (such as Google, Yahoo!, Facebook, and Amazon) and universities.

The basic unit of information in MapReduce is a \((key; value)\) pair. At a high level, computation is the application of a sequence of functions, each taking as input a set of \((key; value)\) pairs and producing as output a new set of \((key; value)\) pairs. Computation proceeds in rounds, and each round consists of three consecutive and independent phases, called map, shuffle, and reduce.

In the map phase, \((key; value)\) pairs are processed individually and a new multiset of \((key; value)\) pairs is produced (emitted). Specifically, there is a set of functions called mappers, one for each \((key; value)\) pair, and each mapper takes one \((key; value)\) pair and outputs a new set (possibly empty) of \((key; value)\) pairs. It is crucial that the map operation is stateless, that is, it operates on one pair at a time, and thus it does not depend on any stored information from previous computations. This allows for easy parallelization as different inputs for the map can be processed by different machines.

In the shuffle phase, the underlying run-time system that implements MapReduce routes all the output \((key; value)\) pairs of the map phase so that all the \((key; value)\) pairs with the same \(key\) end up together in the same machine. This phase occurs automatically, and is totally seamless to the programmer.

In the reduce phase, all the values associated with the same key are processed together. Specifically, there is a set of functions called reducers, one for each \(key\),
and each reducer takes as input a key $k$ and the multiset of values contained in all the $(key; value)$ pairs with $key = k$, and outputs (emits) a new multiset of $(key; value)$ pairs where $key = k$. The reduce phase cannot start until the map phase is completely finished.

The main reason for the success of MapReduce is arguably its simplicity: as the paradigm decouples the programming model from the underlying execution infrastructure, it is the runtime system that manages all the low-level details, thus letting the programmer focus only on the problem, not on the platform. In particular, the framework shields the programmer from all the low-level details of parallel programming such as inter-machine communication, handling machine failures, performing the shuffling step, data distribution, scheduling the program's execution across a set of machines, etc., which are all addressed by the MapReduce environment. The programmer only needs to specify the map and reduce functions; the system-level issues are handled by the underlying implementation.

The canonical example of MapReduce program is counting word frequencies in a text file with a one-round MapReduce protocol.

### 8.1.1 Example: Counting Word Frequencies

**Algorithm 15** Word Count in MapReduce

**Map:**
1: method MAP(docid $a$, doc $d$)
2: for all word $w \in$ doc $d$ do
3: Emit($word \ w$, count 1)

**Reduce:**
1: method REDUCE($word \ w$, counts $[c_1, c_2, \ldots]$)
2: $sum \leftarrow 0$
3: for all count $c \in$ counts $[c_1, c_2, \ldots]$ do
4: $sum \leftarrow sum + c$
5: Emit($word \ w$, count $sum$)

Notice that the above algorithm consists of only one round. However, in general several rounds are needed to solve more complicated problems, as discussed in the next section.

**Exercise 8.1.** Give a MapReduce algorithm that computes the degree of each vertex of a given graph $G = (V, E)$ represented as a list of edges $(u, v)$. How many rounds does it comprise?
Exercise 8.2. Give a MapReduce algorithm that enumerates all the triangles of a given graph \( G = (V, E) \) represented as a list of edges \((u, v)\). How many rounds does it comprise?

8.2 A Model of Computation for MapReduce

Since MapReduce allows the computation to proceed on different parts of the data in parallel, the full power of the framework is realized when the input is so large that it cannot fit into a single machine. Thus, as part of the model we assume that the memory size of each machine is substantially smaller (in particular, sublinear) in the input size \( n \). This eliminates the false efficiency of the trivial one-round algorithm (which maps all the inputs to a single key, and then the reducer for that key performs a standard sequential algorithm to solve the whole problem), and forces the computation to proceed in parallel across the machines of the cluster. Specifically, the model assumes that a single machine has a memory of size \( O(n^{1-\epsilon}) \), where \( n \) is the size of the input and \( \epsilon \) is a constant greater than zero. That is, the memory size of each machine is sublinear in the input size.

For the same reason as before (namely, that MapReduce is aimed for processing large data sets), in order for the model to have practical relevance, the number of available machines is also limited to be substantially smaller (i.e., sublinear) in the input size. Specifically, the model assumes that \( O(n^{1-\epsilon}) \) machines are available for the computation, where \( n \) is the size of the input and \( \epsilon \) is a constant greater than zero. Also, each machine runs in time polynomial in \( n \).

The third and last component of the model is the cost function. As it usually involves the movement of tera- or peta-bytes of data across the network of machines, shuffling turns out to be the time consuming operation. Hence, the cost function is the number of rounds of the computation. One can also define different complexity classes of MapReduce computations: an algorithm \( A \in \mathcal{MRC}^i \) if it runs in \( O(\log^i n) \) rounds.

In particular, for an algorithm to be practical we aim to design MapReduce algorithms that terminate in a constant number of rounds, that is, that belong to class \( \mathcal{MRC}^0 \). The fundamental idea behind this is that in large-scale computations no machine gets to see the whole input during the computation.

Comment: Notice that this model is actually an execution model, which differs from the MapReduce specification model described in the preceding section. The algorithm does not use the number of available machines at the specification
level, and thus the degree of parallelism exposed by an algorithm is decoupled from the one of the system where the computation will be executed.

In order to partially bridge the two models, we can view the reduce phase and the map phase of the subsequent round as a single computation phase. Looking at a MapReduce computation through this lens, in every round each machine performs some computation on the set of \((key; value)\) pairs assigned to it (reduce phase), and then designates which machine each output value should be sent to in the next round (map phase). The shuffle ensures that the data is moved to the right machine, after which the next computation can begin. This way, one can reason about machines as opposed to mappers and reducers, defining what each machine does during each round and specifying which machine each output \((key; value)\) pair goes to. Syntactically, we have that at each round \(i\), the computation performed is \(\phi_i = m_{i+1} \odot r_i(X)\), where \(m_{i+1}\) is map function of round \(i + 1\), \(r_i\) reduce function of round \(i\), \(X\) is a set of \((key; value)\) pairs, and \(\odot\) is the operation of feeding the output of \(r_i(X)\) to \(m_{i+1}\).

### 8.3 MapReduce Algorithms

A ton of MapReduce algorithms have been designed so far, for problems as diverse as linear algebra computations, clustering, matchings, max cover, etc.

We consider two well-studied problems: Densest subgraph and MST.

#### 8.3.1 Densest Subgraph

Let \(G = (V, E)\) be an undirected graph. For a subset \(S \subseteq V\) let the induced edge set be defined as \(E(S) = E \cap S^2\) (i.e., all edges of \(G\) with both endpoints in \(S\)), and let the induced degree of a node \(i \in S\) be defined as \(\deg_S(i) = \{|j\text{ s.t. } (i, j) \in E(S)|\}\). \(\rho(S) = \frac{|E(S)|}{|S|}\) is called density of \(S\), and it is half the average degree of nodes in \(S\).

**Problem:** Given a graph \(G = (V, E)\), find \(V' \subseteq V\) that maximizes \(\rho = \frac{|E(V)|}{|V|}\).

The densest subgraph problem lies at the core of large scale data mining, and as such it and its variants have been intensively studied. A densest subgraph can capture a well-connected relationship among the nodes in the subgraph, and hence can model a community in a large network.

Here we present a \(2(1 + \phi)\)-approximation deterministic MapReduce algorithm, where \(\phi > 0\) is an input parameter, that runs in \(O(\frac{\log n}{\log(1 + \phi)})\) rounds. Starting with the given graph \(G\), the algorithm computes the current density, \(\rho(G)\), and removes all of the nodes (and their incident edges) whose degree is less than \(2(1 + \phi) \cdot \rho(G)\). If the resulting graph is non-empty, then the algorithm recurses on the remaining graph, with node set denoted by \(S\), again computing...
its density and removing all of the nodes whose degree is lower than the specified threshold; we denote these nodes by $A(S)$. Then, the node set reduces to $S \setminus A(S)$, and the recursion continues in the same way. Algorithm 8.3.1 presents the complete description.

Algorithm 16 Approximate Densest Subgraph

Require: $G = (V, E)$ and $\phi > 0$
1: $\bar{S} \leftarrow V$
2: $S \leftarrow V$
3: while $S \neq \emptyset$ do
4: $A(S) \leftarrow \{i \in S \text{ s.t. } \deg_S(i) \leq 2(1 + \phi)\rho(S)\}$
5: $S \leftarrow S \setminus A(S)$
6: if $\rho(S) > \rho(\bar{S})$ then
7: $\bar{S} \leftarrow S$
8: return $\bar{S}$

Theorem 8.1. For any $\phi > 0$, Algorithm 8.3.1 returns a $2(1 + \phi)$-approximate solution for the densest subgraph problem $O(\frac{\log n}{\log(1+\phi)})$ MapReduce rounds.

Proof. We begin by proving the correctness of the algorithm, that is, Algorithm 8.3.1 returns a $2(1 + \phi)$-approximation to the densest subgraph problem. First of all, notice that at least one node must be removed in every iteration of the while loop, that is, there will always be a node $i$ in line 4 of the algorithm. In order to argue this, observe that the average degree of nodes in $S$ is

$$\frac{\sum_{i \in S} \deg_S(i)}{|S|} = \frac{2|E(S)|}{|S|} = 2\rho(S).$$

Since it is not possible that all nodes in $S$ have degree bigger than their average, this proves that at least one node must be removed in every pass, hence the algorithm terminates (in at most $n$ rounds, but soon we will see it needs actually much less than $n$ rounds). Now, consider the first time in the pass when a node $i$ from the optimal solution $S^*$ is removed. (This moment is guaranteed to exist, since as we just argued $S$ eventually becomes empty.) Clearly $S^* \subseteq S$ (before removing nodes from $S$). Let $i \in A(S) \cap S^*$. If $\rho(S^*) \leq \deg_S(i)$ holds, we have

$$\rho(S^*) \leq \deg_S(i) \leq \deg_S(i) \leq 2(1 + \phi)\rho(S) \quad i \in A(S),$$
which gives the claimed quality of the output solution. It is easy to see that
\( \rho(S^*) \leq \deg_{S^*}(i) \) holds: by the optimality of \( S^* \), for each \( i \in S^* \) we have

\[
\rho(S^*) \geq \rho(S^* \setminus \{i\}) = \frac{|E(S^*)| - \deg_{S^*}(i)}{|S^*| - 1}
\]

\[
\rho(S^*)(|S^*| - 1) \geq |E(S^*)| - \deg_{S^*}(i)
\]

\[
\rho(|S^*|)(|S^*|) - |E(S^*)| - \rho(S^*) \geq -\deg_{S^*}(i)
\]

\[
\rho(S^*) \leq \deg_{S^*}(i)
\]

We finally note that the algorithm returns the set \( \tilde{S} \) which is the set with the largest \( \rho \) encountered (lines 6 and 7 of the Algorithm). Thus, its \( \rho \) value will be at least as large as \( S \).

We now prove the running time of the algorithm. Observe that at each round we have

\[
2|E(S)| = \sum_{i \in S} \deg_S(i)
\]

\[
= \sum_{i \in A(S)} \deg_S(i) + \sum_{i \in S \setminus A(S)} \deg_S(i)
\]

\[
> (|S| - |A(S)|)2(1 + \phi)\rho(S)
\]

\[
= (|S| - |A(S)|)2(1 + \phi)\frac{|E(S)|}{|S|},
\]

where the inequality follows by considering only the second addendum. Thus,

\[|A(S)| > \frac{\phi}{1 + \phi}|S|.
\]

Equivalently,

\[|S \setminus A(S)| < \frac{1}{1 + \phi}|S|.
\]

Therefore, the cardinality of the remaining set \( S \) decreases by a factor at least \( 1/(1 + \phi) \) during each round. Hence, the algorithm terminates in \( O(\log_{1+\phi} n) = O\left(\frac{\log n}{\log(1+\phi)}\right) \) rounds.

Finally, it is easy to see that the constraint on the memory used by each machine is never violated.

**Exercise 8.3.** Explain how the map and reduce steps for the densest subgraph problem are implemented. Why does the memory constraint used by each machine never violated?
8.3.2 Minimum Spanning Tree (MST)

Through the minimum spanning tree problem, we will illustrate a general technique for the design of MapReduce algorithms with good performance, i.e., run in a small number of rounds. The technique is called filtering, and it is one of the very few known general techniques for the design of efficient MapReduce algorithms, in particular for problems on graphs. Informally, this technique works as follows. First, the algorithm drops, or filters, part of the input with the goal of reducing the problem size (possibly so that the result is small enough to fit into a single machine’s memory). Second, the algorithm performs some computation on this smaller input, and possibly repeats it. The technical challenge, clearly, is to choose enough information to drop but still enough to compute the desired (maybe approximate) solution.

Notation: Let $G = (V, E)$ be an undirected graph, where $n = |V|$ and $m = |E|$. We say that $G$ is $c$-dense if $m = n^{1+c}$ for some constant $c$ with $0 < c \leq 1$.\footnote{Typical large graphs have $c \in [0.08, 0.5]$.}

In this subsection we will assume that the memory of each machine has size $\eta = O(n^{1+\epsilon})$, for some constant $0 < \epsilon < c$. (Note that this complies with the requirements of the adopted model of computation as the input size here is $m = n^{1+c}$ and thus the memory size of any machine is $O(n^{1+\epsilon}) = O(m^{1-\epsilon'})$ where $\epsilon' = (c - \epsilon)/(c + 1)$ is a constant greater than zero, as desired.) We assume that the number of available machines is $O(m/\eta) = O(n^{c-\epsilon})$. (Again, it is easy to see that this is sublinear in the input size as prescribed by the model.)

In this subsection we consider the problem of computing a spanning tree (or spanning forest, if the graph is not connected) of minimum cost in a weighted graph $G$. (The very same algorithm can also be used to find the connected components of $G$.) We will assume that the graph is $c$-dense. We will present a randomized MapReduce algorithm, called MST$(V, E)$, that with high probability (whp) returns a minimum spanning tree in $O(1)$ rounds.

The algorithm randomly partitions the edges of $G$ into subsets of size $\eta$, and sends each subset to a distinct machine. Then, each machine independently discards any edge that is guaranteed not to be part of any minimum spanning tree (filtering phase). This is repeated until the resulting graph fits into the memory of a single machine; this graph is then sent to one machine that on its own computes a MST of the resulting graph. The pseudocode of the algorithm is shown in Algorithm\footnote{Algorithm 17}.

Now we analyze the correctness and performance of this algorithm in our model of computation.
Algorithm 17 MST($V, E$)

1: if $|E| < \eta$ then
2: \hspace{1em} compute $T^* = MST(V, E)$
3: \hspace{1em} return $T^*$
4: \hspace{1em} $\ell \leftarrow \Theta(|E|/\eta)$ \hspace{1em} / \hspace{1em} $\ell =$ number of machines to be used in the current round
5: \hspace{1em} partition $E$ into $E_1, E_2, \ldots, E_\ell$ where $|E_i| = \Theta(\eta)$ using a universal hash function $h : E \rightarrow \{1, 2, \ldots, \ell\}$
6: \hspace{1em} in parallel, compute $T_i$, the minimum spanning forest on $G = (V, E_i)$
7: \hspace{1em} return $MST(V, \bigcup_i T_i)$

Theorem 8.2. Algorithm MST($V, E$) returns a minimum spanning tree of a $c$-dense graph in $O(1)$ MapReduce rounds.

Proof. We have to prove the correctness and provide an upper bound on the round complexity of the algorithm. To claim correctness we have to show that the algorithm returns a minimum spanning tree of the input graph $G$. Clearly, the algorithm returns a spanning tree of $G$. We prove that it is a spanning tree of minimum cost by arguing that no edge pruned before the last round is part of a spanning tree of $G$ of cost smaller than the cost of the spanning tree returned by algorithm MST($V, E$). This is ensured by the cycle property of minimum spanning trees.

**Cycle property of minimum spanning trees:** Let $T$ be a minimum spanning tree of a weighted graph $G$. Let $a$ be an edge of $G$ that is not in $T$, and let $C$ be the cycle formed by $a$ with $T$. Then, for every edge $b$ of $C$, weight($b$) $\leq$ weight($a$). Thus, each edge thrown out during the algorithm was part of a cycle, and had weight no smaller than all other edges of the cycle. Therefore, any such edge cannot appear in the MST. Hence, the edges computed at the end of the algorithm form a minimum spanning tree of $G$.

We now analyze the performance of the algorithm. Specifically, we will show that (1) w.h.p. the memory constraint of each machine is never violated, and (2) the algorithm terminates in $O(1)$ MapReduce rounds. We first prove (1). Since the partition of the edges among the machines is done randomly, (1) is ensured by a simple application of a Chernoff bound:

**Chernoff bound:** Let $X_1, X_2, \ldots, X_j$ be independent random variables, with $0 \leq X_i \leq 1$. Let $X = \sum_{i=1}^j X_i$ and $\mu = E[X]$. Then,

$$\Pr[|X - \mu| \geq \alpha] \leq 2e^{-2\alpha^2/j}.$$ 

The application of this bound for the first round is as follows (for subsequent rounds it is similar):
8.3. MAPREDUCE ALGORITHMS

- \(X_i = 1\) if edge \(i\) is sent to machine \(y\), 0 otherwise
- \(j = n^{1+\epsilon}\)
- \(X = \text{number of edges on machine } y\)
- \(\mu = E[X] = E\left[\sum_{i=1}^{n^{1+\epsilon}} X_i\right] = \sum_{i=1}^{n^{1+\epsilon}} \frac{1}{n^c} = \frac{n^{1+\epsilon}}{n^c} = n^{1+\epsilon}\) (by linearity of expectation)
- \(\alpha = n^{1+\epsilon}\)

Applying the Chernoff bound,

\[
\Pr[X > 2n^{1+\epsilon}] \leq \Pr[|X - n^{1+\epsilon}| > n^{1+\epsilon}]
\leq 2e^{-2(n^{1+\epsilon})^2/n^{1+\epsilon}}
= 2e^{-2n^{1+2\epsilon-c}},
\]

that is, w.h.p. any machine \(y\) receives no more than \(2n^{1+\epsilon}\) edges. The claim that all the machines get no more than \(2n^{1+\epsilon}\) edges follows by applying the union bound to the \(O(n^{c-\epsilon})\) machines.

It remains to prove (2), that is, that the algorithm terminates in \(O(1)\) MapReduce rounds. In fact, by construction every iteration reduces the input size by a factor of \(\Theta(n^{c})\), as each machine receives as input \(\Theta(n^{1+\epsilon})\) edges and returns at most \(n - 1\) edges. Therefore, after \(\lceil c/\epsilon \rceil - 1 = O(1)\) iterations the input is small enough (\(\Theta(n^{1+\epsilon})\) edges) to fit into a single machine, which will return a MST of the original graph \(G\).

\[\Box\]

Exercise 8.4. Consider the problem of finding a maximal matching of a graph. Give an example that shows that the same strategy used for MST does not work for this problem, i.e., show that it is impossible to build a maximal matching using only the non-filtered edges.

Exercise 8.5. The prefix sum problem asks, given a list \(a_1, a_2, \ldots, a_n\) of \(n\) values, to return a list \(b_1, b_2, \ldots, b_n\) where \(b_i = \sum_{j=1}^{i} a_j\) for each \(i \in \{1, 2, \ldots, n\}\). Describe and analyze an algorithm for prefix sum assuming that the input is given in pairs \((i; a_i)\).
Chapter 9

$k$-machine Model and Algorithms

9.1 Introduction

Till now we have focused on studying fundamental distributed network algorithms. In this chapter, we will take a different view and study distributed algorithms for graphs. What are the differences between the two?

Distributed network algorithms have been studied for over the last three decades mainly in the context of distributed communication networks (e.g., the Internet, peer-to-peer networks, and ad hoc wireless networks), where they crucially enable fundamental network operations such as broadcast, multicast, routing, search etc. (We have till now focused on distributed algorithms from this perspective.) At its core, as we have seen, distributed network algorithms are graph algorithms, but there is a big difference in the way these algorithms are modeled, designed, and analyzed compared to the centralized setting. As we have assumed till now, each node (which represents a processor in a communication network) computes in a decentralized and localized manner; nodes also can communicate with their neighbors by exchanging messages. We have studied distributed algorithms for important graph problems such as spanning tree, shortest paths etc., which are widely used in modern communication networks. In distributed computation, communication is at least as important as computation within a node; in particular, communication between nodes is typically the costly operation, and dominates the overall cost of the algorithm. On the other hand, the emergence of “Big Data” over the last decade has led to many new computing platforms for distributed processing of large-scale data, exemplified by MapReduce and Hadoop, and more recently systems such as
Pregel, Giraph, GPS, GraphLab, Spark, etc. In these platforms, the data—which is typically too large to fit into a single machine—is distributed across a group of machines that are connected via a communication network, and the machines jointly process the data in a distributed fashion.

In the last chapter, we studied the MapReduce model which is a model for distributed processing of large-scale data. In this chapter, we will study another model called the $k$-machine model, a general model that captures distributed computation of data over a network of machines. MapReduce (developed at Google) has become a very successful distributed computing platform for a wide variety of large-scale computing applications and also has been used for processing graphs. However, as pointed out by the developers of Pregel (which was also developed at Google), MapReduce may sometimes be ill-suited for implementing distributed algorithms, especially graph algorithms. One reason is that many graph algorithms may take a lot of (MapReduce) communication rounds which might be too costly. Indeed, the MapReduce model is stateless, i.e., the data is not really associated with any particular machine and can be moved around (from one machine to another) from one round to the next. This can lead to a lot of communication per round; hence MapReduce algorithms typically work best when there are very few rounds, say, $O(1)$ rounds.

On the other hand, graph algorithms are better suited to a message-passing distributed computing model and this is the main design principle behind Pregel (and other systems that followed it such as Giraph and Spark). Hence, the algorithms and theory for the message-passing distributed computing model can be leveraged to study distributed graph processing systems.

In this chapter, we introduce the $k$-machine model as a model for distributed processing of data and present algorithms for several problems. We give a high-level overview of the model and the ideas behind it; more details are in Section 9.2.

The $k$-machine model consists of a point-to-point communication network of $k$ machines interconnected by bandwidth-restricted links; the machines communicate by message passing over the links. The network is used to process some data (typically large) that is partitioned across the machines in a balanced fashion. The data can be a (large) graph or other types of data. For example, the data can be a set of $n$ numbers partitioned across the machines with each machine getting $n/k$ numbers. Or the input can be an arbitrary $n$-node graph $G$. Vertices and edges of $G$ are partitioned across the machines in a (approximately) balanced manner; in particular, one can assume that the vertices (and their incident edges) are partitioned in a random fashion, which is a common implementation in many real-world systems. As usual, we assume that the distributed computation proceeds in a sequence of rounds. In a round, each machine does some local
computation and can send/receive messages. We use the receive-compute-model that we had assumed throughout. Local computation within a machine is considered free, while communicating messages between the machines is the costly operation. This assumption is reasonable in the context of large-scale data — indeed, typically in practice, even assuming the links have a bandwidth of order of gigabytes of data per second, the amount of data that have been to be communicated can be in order of tera or peta bytes which generally dominates the overall computation cost.

Let’s compare and contrast the $k$-machine model with the usual message-passing distributed computing model that we have studied till now, i.e., the synchronous CONGEST model operating on a distributed network. As we know, in this model, each node in the network represents a processor (or a computing element) and the edges represent the communication links between the nodes. Let’s call this the vertex-centric model, since here the vertices do the computation and send/receive messages. One can obtain algorithms in the $k$-machine model by simulating vertex-centric algorithms on the $k$-machine model (in fact, this is the main method that we will use to design $k$-machine algorithms in this Chapter). In contrast, in the $k$-machine model the vertices of the input graph don’t really perform any computation and the simulation of a vertex is done by the machine that holds the vertex (although, in principle, any machine can do the simulation). On the other hand, the $k$-machine model itself is simply a distributed network whose topology is a $k$-clique. This clique network operates as a synchronous CONGEST model, where each link (there are $\binom{k}{2}$ links) has a bandwidth constraint.

A fundamental issue that we would like to study is how the number of communication rounds scales with the number of machines used: more precisely, if we use $k$ machines, does the rounds scale linearly (or even super-linearly) in $k$?

9.2 Model

We consider a network of $k > 1$ (distinct) machines $N = \{p_1, \ldots, p_k\}$ that are pairwise interconnected by bidirectional point-to-point communication links — henceforth called the $k$-machine model. Each machine executes an instance of a

---

1Many real-world systems such as Pregel, Giraph, and Spark use vertex-centric algorithms, where vertices are the objects which do computation and send/receive messages to their respective neighbors, much like our standard message-passing model. However, note that one can also directly design algorithms for the $k$-machine model, and this can be more efficient than simulating corresponding algorithms designed for the vertex-centric model; we won’t focus on this aspect in this Chapter.
distributed algorithm $A$. The computation advances in *synchronous* rounds where, in each round, machines can exchange messages over their communication links.

Each link is assumed to have a bandwidth of $B$, i.e., $B$ bits can be transmitted over the link in one round. As usual, for convenience, we will assume throughout that bandwidth $B = O(\log n)$, where $n$ is the size of the input data; in any case, it is easy to rewrite our upper bounds to scale in terms of parameter $B$. Note that, as before, machines have no other means of communication and do not share any memory; all communication is via message-passing. As we have assumed before, local computation within a machine is considered free, while communicating messages between the machines is the costly operation. This assumption is also reasonable in the context of large-scale data. Indeed, typically in practice, even assuming the links have a bandwidth of order of gigabytes of data per second, the amount of data that have been to be communicated can be in order of tera or peta bytes which generally dominates the overall computation cost.

Although the $k$-machine model can be used to distributively solve any problem, say, e.g., sorting (Exercise 9.1), in this Chapter, we are mainly interested in solving graph problems where we are given an input graph $G$ of $n$ vertices (assume that each vertex has a unique label) and $m$ edges. To avoid trivialities, we will assume that $n \geq k$ (typically, in practice, $n \gg k$). Unless otherwise stated, we will consider $G$ to be undirected. Initially, the entire graph $G$ is not known by a single machine, but rather partitioned among the $k$ machines in a “balanced” fashion, i.e., the nodes and/or edges of $G$ must be partitioned approximately evenly among the machines. We will assume a vertex-partition model, where vertices (and their incident edges) are partitioned across machines. In particular, we will assume the random (vertex) partition, i.e., vertices (and its incident edges) of the input graph are assigned randomly to machines\footnote{This is the typical way that many real systems (e.g., Pregel) partition the input graph among the machines; it is simple and and easy to accomplish, e.g., via hashing.}.

Formally, in the random vertex partition (RVP) model, each vertex of $G$ is assigned independently and randomly to one of the $k$ machines. If a vertex $v$ is assigned to machine $p_i$ we call $p_i$ the home machine of $v$. Note that when a vertex is assigned to a machine, *all its incident edges* are assigned to that machine as well; i.e., the home machine will know the labels of neighbors of that vertex as well as the identity of the home machines of the neighboring vertices. A convenient way to implement the above assignment is via *hashing*: each vertex (label) is hashed to one of the $k$ machines. Hence, if a machine knows a vertex
9.3 DESIGNING ALGORITHMS FOR THE K-MACHINE MODEL

label, it also knows where it is hashed to.

Depending on the problem \( \mathcal{P} \), the vertices and/or edges of \( G \) have labels chosen from a set of polynomial (in \( n \)) size. Eventually, each machine \( p_i \) (\( 1 \leq i \leq k \)) must output a variable \( o_i \) (which may depend on the set of vertices assigned to machine \( p_i \)) and the output \( o = (o_1, \ldots, o_k) \) must satisfy certain feasibility conditions w.r.t. problem \( \mathcal{P} \). For example, when considering the minimum spanning tree (MST) problem, each \( o_i \) corresponds to a set of edges, and the edges in the union of the sets \( o_i \) must form an MST of the input graph \( G \). Note that it is not required that a particular edge should be output by a specific machine (although, one can design an algorithm where each machine outputs the MST edges incident to the vertices assigned to that machine — note that this is a natural generalization of the analogous assumption in the standard distributed message passing model, where each vertex knows which of its incident edges belong to the MST.)

We say that algorithm \( A \) solves problem \( \mathcal{P} \) if \( A \), for each input instance of the problem gives an output that is feasible for \( \mathcal{P} \). The round complexity of \( A \) is the maximum number of communication rounds until termination, over all input instances. The round complexity captures the communication cost of the algorithm, since links can transmit only a limited amount of bits per round. Hence, to minimize communication — a key goal — we would prefer algorithms with low round complexity.

9.3 Designing algorithms for the \( k \)-machine model

As mentioned earlier, we will use a fairly straightforward methodology to design algorithms in the \( k \)-machine model. Although this technique does not always give the best algorithm, it gives an algorithm that is reasonably good. In particular, it will be better than the naive algorithm which simply sends all the data to one machine which then locally computes the solution. For graph problems, thus one can trivially get a \( O(m+n) \) round algorithm, since all the vertices and edges can be sent to one machine and since that machine has only \( k - 1 \) links, this takes so many rounds.

Given a problem \( \mathcal{P} \), the methodology shows how fast algorithms for solving \( \mathcal{P} \) in the \( k \)-machine model can be designed by leveraging distributed algorithms for \( \mathcal{P} \) in the standard CONGEST message-passing distributed computing model, i.e., the vertex-centric model. The methodology is summarized in the Conversion Theorem. Thus using this theorem one can convert distributed graph algorithms in the vertex-centric model to apply in the \( k \)-machine model.

We note that fast distributed algorithms in the standard model may not
directly imply fast algorithms in the $k$-machine model. The main reason for this is that while in the standard vertex-centric model, the topology is arbitrary and not a complete network (i.e., a clique), whereas the $k$-machine model is a clique. To achieve faster algorithms, we consider vertex-centric algorithms in an intermediate clique model and then show two ways — parts (a) and (b) respectively of the Conversion Theorem — to efficiently convert algorithms in the clique model to the $k$-machine model. Part (b) applies to converting distributed algorithms (in the clique model) that only uses broadcast (note that in a broadcast algorithm, in a round, every node, sends the same message to a subset (or all) of its neighbors; the rest of the neighbors don’t get any message), while part (a) applies to any algorithm. Part (a) will sometimes give better time bounds compared to part (b) and vice versa — this depends on the problem at hand and the type of distributed algorithm considered, as well as on the graph parameters. (The latter can be especially useful in applications where we might have some information on the graph parameters/topology as explained below.) Using this theorem, one can design algorithms for various graph problems, e.g., PageRank, minimum spanning tree (MST), connectivity, spanning tree (ST) verification, shortest paths, Triangle enumeration etc. Problems such as PageRank, MST, and connectivity, graph covering etc. can be solved in $\tilde{O}(n/k)$ time (note that $\tilde{O}$ notation hides a polylog($n$) multiplicative and additive factor); this shows that one can achieve almost linear (in $k$) scaling. For graph connectivity, BFS tree construction, and ST verification, can be solved in $\tilde{O}(\min(n/k, m/k^2 + D\Delta/k))$ bound — note that the second part of the above bound may be better in some cases, e.g., if the graph is sparse (i.e., $m = O(n)$) and $D$ and $\Delta$ are small (e.g., bounded by $O(\log n)$) — then we get a bound of $\tilde{O}(n/k^2)$.

9.3.1 A Mapping Lemma

We first prove a “mapping” lemma for the random vertex partition model, which is needed for the Conversion theorem (cf. Theorem 9.1). The Mapping Lemma gives a bound on the number of vertices and edges assigned to each machine as well as the number of edges “mapped” (defined below) to any link of the $k$-clique. Consider an input graph $G = (V,E)$ that is partitioned (according to the random vertex partition model) among the $k$ machines in $N = \{p_1, \ldots, p_k\}$ of the network. Let $|V| = n$ and $|E| = m$. We will assume throughout that $n \geq k$.

We say that a vertex $v$ of $G$ is mapped to a machine $h$ of $N$ if $v$ is assigned to $N$, i.e., $h$ is the home machine of $v$ (cf. Section 9.2). We say that an edge $e = (u, v)$ of $G$ is mapped to a link $(p_i, p_j)$ of the $k$-machine network, if $u$ is mapped to $p_i$ and $v$ is mapped to $v_j$ or vice versa. The following Mapping Lemma gives a concentration bound (i.e., a high probability bound) on the number of edges
mapped to any link of the network. The statement of the Mapping lemma is what we need for the Conversion Theorem; the proof can be skipped for first reading.

**Lemma 9.1** (Mapping Lemma). Let an $n$-node, $m$-edge graph $G$ be partitioned among the $k$ machines in $N = \{p_1, \ldots, p_k\}$, according to the random vertex partition model (assume $n \geq k$). Then with probability at least $1 - 1/n^\alpha$, where $\alpha > 1$ is an arbitrary fixed constant, the following bounds hold:

1. The number of vertices of $G$ mapped to any machine is $\tilde{O}(n/k)$.
2. The number of edges of $G$ mapped to any link of the network is $\tilde{O}(m/k^2 + \Delta/k)$, where $\Delta$ is the maximum node degree of $G$.

**Proof.** (1) This follows easily from a direct Chernoff bound application. Since each vertex of $G$ is mapped independently and uniformly to the set of $k$ machines, the expected number of vertices mapped to a machine (by linearity of expectation) is $n/k$. The concentration follows from a Chernoff bound (see Appendix).

(2) We first note that we cannot directly apply a Chernoff bound to show concentration on the number of edges mapped, as these are not independently distributed. We show the bound in two steps:

1. (a) We first show a concentration bound on the total degree of the vertices assigned to any machine;
2. (b) then we bound the number of edges assigned to any link.

To show (a) we use Bernstein's inequality (see Appendix). Fix a machine $p$. Let random variable $X^p_i$ be defined as follows: $X^p_i = d(v_i)$ if vertex $v_i$ is assigned to machine $p$, otherwise $X^p_i = 0$. Let $X^p = \sum_{i=1}^{n} X^p_i$ denote the total degree of the vertices assigned to machine $p$; in other words, it is the total number of edges that have at least one endvertex assigned to machine $p$. We have $E[X^p_i] = d(v_i)/k$ and $E[X^p] = \sum_{i=1}^{n} E[X^p_i] = \sum_{i=1}^{n} d(v_i)/k = 2m/k$. Furthermore, $\text{Var}(X^p) = E[(X^p)^2] - E[X^p]^2 = \frac{1}{k}(d(v_i))^2 - \frac{(d(v_i))^2}{k} = \frac{(d(v_i))^2}{k}(1 - 1/k)$ and hence $\text{Var}(X^p) = \sum_{i=1}^{n} \text{Var}(X^p_i) = \frac{1}{k}(1 - 1/k) \sum_{i=1}^{n} (d(v_i))^2 \leq \frac{1}{k}(1 - 1/k) \sum_{i=1}^{n} \Delta d(v_i) = \frac{1}{k}(1 - 1/k) \Delta m$.

Using Bernstein's inequality, we have (for some $\epsilon > 0$):

$$
\Pr(X^p > E[X^p] + \epsilon) \leq e^{-\frac{\epsilon^2}{\epsilon^2}}
$$

where $b = \max_{1 \leq i \leq n} |X^p_i - E[X^p_i]|$. Now, $|X^p_i - E[X^p_i]| \leq d(v_i)(1 - 1/k) \leq \Delta(1 - 1/k) = b$. 

103
Let $\gamma > 0$ and let $A$ be the event that $X^p > 2m/k + \gamma(2m/k + \Delta)$. Hence, for any $\gamma > 0$ and letting $t = \gamma(2m/k + \Delta)$, we have:

$$
\Pr(A) \leq e^{-\frac{t^2}{2(k/k+1/\Delta)\Delta(1-1/k)}} \leq e^{-\frac{t^2}{2(k/k+1/\Delta)\Delta}} \leq e^{-\frac{t^2}{2(\Delta+1/k+\Delta)}} \leq e^{-\gamma^2 \Theta(\frac{1}{\Phi} + \frac{1}{k} + 1)} = O(1/n^{3\alpha}),
$$

if $\gamma = \Theta(\alpha \sqrt{\log n})$.

The above tail bound applies to a single machine $p$; applying a union bound over all the $k$ machines, we have $X^p = \tilde{O}(m/k + \Delta)$ whp for every machine $p \in N$.

We now show (b) which will complete the proof of Part 2. Fix a link $\ell = (p, q)$ of the $k$-machine network. Let $X^p$ (resp. $X^q$) be defined as before, i.e., the total number of edges of $G$ with at least one end-vertex assigned to $p$ (resp. $q$). Note that each such edge mapped to $p$ has probability of $1/(k-1)$ of being mapped to link $\ell$ (independently of other edges mapped to $p$). A similar statement holds for edges mapped to $q$ as well. Let r.v. $Z^p_\ell$ (resp. $Z^q_\ell$) denote the number of edges, among those mapped to $p$ (resp. $q$), that are mapped to link $\ell$; the total number of edges mapped to $\ell$ is bounded by $Z^p_\ell + Z^q_\ell$. We have, $E[Z^p_\ell | X^p] \leq \lfloor \frac{X^p}{k^2} \rfloor$ and similarly $E[Z^q_\ell | X^q] \leq \lfloor \frac{X^q}{k^2} \rfloor$. We next show that $Z^p_\ell$ and $Z^q_\ell$ are both bounded by $\tilde{O}(m/k^2 + \Delta/k)$ whp which will complete the proof. We focus on $Z^p_\ell$; (the case of $Z^q_\ell$ is similar). By proof of (a), $X^p < \Phi$ with probability at least $1 - 1/n^{2\alpha}$, where $\Phi = c \log(n(m/k + \Delta))$ for some sufficiently large constant $c > 0$, depending on constant $\alpha$. Observe that an edge that has one endpoint mapped to $p$ is mapped to link $\ell$ independently. Hence we can apply a standard Chernoff bound: $\Pr(Z^p_\ell < c' \Phi/k + 1) \leq \Pr(X^p > \Phi) + \Pr(Z^p_\ell < c' \Phi/k + 1) | X^p < \Phi) \leq 1/n^{2\alpha} + 2^{-c' \Phi/k + 1} \leq 1/n^{2\alpha} + 2^{-3 \log n + \Omega(1)} = O(1/n^{2\alpha})$, for a sufficiently large constant $c' > 0$.

Thus, with probability $1 - 1/n^{2\alpha}$, the number of edges mapped to link $\ell$ is $c' \Phi/k = \tilde{O}(m/k^2 + \Delta/k)$. Applying a union bound over all $k(k-1)/2$ links, yields the result. \hfill \Box

### 9.3.2 The Conversion Theorem
In this section, we give a general Conversion theorem that enables us to convert vertex-centric algorithms to work efficiently in the $k$-machine model. The Conversion theorem allows one to use distributed algorithms that leverage direct communication between nodes, even when such an edge is not part of the input
9.3. DESIGNING ALGORITHMS FOR THE K-MACHINE MODEL

In this section, we consider algorithms for the $k$-machine model, where each machine is a complete graph. More specifically, we can translate any distributed algorithm that works in the following clique model to the $k$-machine model.

**The Clique Model.** Consider a complete $n$-node network $C$ and a spanning subgraph $G$ of $C$ determined by a set of (possibly weighted) edges $E(G)$. The nodes of $C$ execute a distributed algorithm and each node $u$ is aware of the edges that are incident to $u$ in $G$. Each node can send a message of at most $B = O(\log n)$ bits over each incident link per round. For a graph problem $P$, we are interested in distributed algorithms that run on the network $C$ and, given input graph $G$, compute a feasible solution of $P$. In addition to round complexity (the number of rounds in the worst case), we are interested in the message complexity of an algorithm in this model which is the number of messages (in the worst case) sent over all links. Additionally, we are also interested in communication degree complexity which is the maximum number of messages sent or received by any node in any round; i.e., it is the minimum integer $M'$ such that every node sends a message to at most $M'$ other nodes in each round. Note that we can simulate any vertex-centric algorithm running on a network $G$ of an arbitrary topology that uses messages of $O(\log n)$ size in the clique model by simply restricting the communication to edges in $E(G) \subseteq E(C)$. In this case, the time and message complexities remain the same while the communication degree complexity can be bounded by the maximum degree of $G$. We say that an algorithm is a broadcast algorithm if, in every round and for every node $u$, it holds that $u$ broadcasts the same message to a subset (or all) of its neighbors (it does not send any message at all to rest of the neighbors). We define the broadcast complexity of an algorithm as the number of times nodes broadcast messages.

**Terminology.** We next introduce some terminology. For any $0 \leq \epsilon \leq 1$, we say that an algorithm has $\epsilon$-error if, for any input, it outputs the correct answer with probability at least $1 - \epsilon$, where the probability is over the random partition and the random bit strings used by the algorithm (in case it is randomized).

For any $n > 0$ and function $T(n)$, we say that an algorithm $\mathcal{A}$ terminates in $O(T(n))$ rounds if, for any $n$-node input graph $G$, $\mathcal{A}$ always terminates in $O(T(n))$ rounds, regardless of the choice of the (random) input partition. For any $n$ and problem $\mathcal{P}$ on $n$ node graphs, we let the round complexity of solving $\mathcal{P}$ with $\epsilon$ error probability in the $k$-machine model, denoted by $T^k_\epsilon(\mathcal{P})$, be the minimum $T(n)$ such that there exists an $\epsilon$-error protocol that solves $\mathcal{P}$ and terminates in $T(n)$ rounds. We will usually use $\epsilon = 1/n$, which will imply high probability algorithms, i.e., succeeding with probability at least $1 - 1/n$. In this case, we will sometimes just omit $\epsilon$ and simply say the time bound applies “with high probability.” We always use $k$ for the number of machines, $n$ for the number of nodes in the input graph, and $m$ for the number of edges in the input graph,
Δ to denote the maximum degree of any node in the input graph, and D for the diameter of the input graph.

**Theorem 9.1** (Conversion Theorem). Suppose that there is an ϵ-error algorithm $A_C$ that solves problem P in time $T_C(n) \in \tilde{O}(n)$ in the clique model, for any n-node input graph. Then there exists an ϵ-error algorithm $A$ that solves P in the k-machine model satisfying the following round complexity bounds with high probability:

(a) If $A_C$ uses point-to-point communication with message complexity $M$ and communication degree complexity $\Delta'$, then $A$ runs in $\tilde{O}\left(\frac{M}{k^2} + T_C(n)\left\lceil \frac{\Delta}{k} \right\rceil\right)$ rounds.

(b) If $A_C$ is a broadcast algorithm with broadcast complexity $R$, then $A$ takes $\tilde{O}\left(\frac{R}{k} + T_C(n)\right)$ rounds.

**Proof idea.** We present the main ideas of the proof of Theorem 9.1. To obtain algorithm $A$ for the k-machine model, each machine locally simulates the execution of $A_C$ at each hosted vertex. If algorithm $A_C$ requires a message to be sent from a node $u_1 \in C$ hosted at machine $p_1$ to some node $u_2 \in C$ hosted at $p_2$, then $p_1$ sends this message directly to $p_2$ via the links of the network $N$.

We will now bound the necessary number of rounds for simulating one round of algorithm $A_C$ in the k-machine model: We observe that we can bound the number of messages sent in a round of $A_C$ through each machine link using Lemma 9.1(2). Let $G_i$ be the graph that captures the communication happening in round $i$ of $A_C$, i.e., there exists an edge $(u, v) \in E(G_i)$ if $u$ and $v$ communicated in round $i$. By Lemma 9.1(2), each communication link of $N$ is mapped to at most $\tilde{O}\left(\frac{|E(G_i)|}{k^2} + \Delta_i/k\right)$ edges of $G_i$ (whp), where $\Delta_i$ is the maximum degree of $G_i$. Summing up over all $T_C(n)$ rounds yields Part (a).

For (b), we modify the previous simulation to simulate a broadcast algorithm $A_C$ in our k-machine model: Suppose that in the $i^{th}$ round, a node $u$ on machine $p_1$ broadcasts a message to nodes $v_1, \ldots, v_j$ on machine $p_2$. We can simulate this round of $A_C$, by letting machine $p_1$ send only one message to $p_2$ and machine $p_2$ will pretend that this message is sent from $u_1$ to all nodes belonging to $p_2$. Recalling Lemma 9.1(a), the number of nodes contributing to $B_i$ broadcasts assigned to a single machine is $\tilde{O}\left([B_i/k]\right)$ w.h.p; appropriately summing up over all $T_C(n)$ rounds, yields the result.

**Proof.** Consider any n-node input graph $G$ with $m$ edges and suppose that nodes in $G$ are assigned to the $k$ machines of the network $N$ according to the vertex partitioning process (cf. Section 9.2).
9.3. DESIGNING ALGORITHMS FOR THE K-MACHINE MODEL

We now describe how to obtain algorithm $A$ for the $k$-machine model from the clique model algorithm $A_C$: Each machine locally simulates the execution of $A_C$ at each hosted vertex. First of all, we only need to consider inter-machine communication, since local computation at each machine happens instantaneously at zero cost. If algorithm $A_C$ requires a message to be sent from a node $u_1 \in C$ hosted at machine $p_1$ to some node $u_2 \in C$ hosted at $p_2$, then $p_1$ sends this message directly to $p_2$ via the links of the network $N$. (Recall that a machine $p_1$ knows the hosting machines of all endpoints of all edges (in $G$) that are incident to a node hosted at $p_1$.) Moreover, $p_1$ adds a header containing the IDs of $u_1$ and $u_2$ to ensure that $p_2$ can correctly deliver the message to the simulation of $A_C$ at $u_2$.

Proof of (a): We will bound the number of messages sent in each round through each link using Lemma 9.1(2). Let $G_i$ be the graph whose node set is the same as the input graph (as well as the clique model), and there is an edge between nodes $u$ and $v$ if and only if a message is sent between $u$ and $v$ in round $i$ of the algorithm; in other words, $G_i$ captures the communications happening in round $i$. From Lemma 9.1(2), we know that (w.h.p.) each communication link of $N$ is mapped to at most $\tilde{O}(|E(G_i)|/k^2 + \Delta_i/k)$ edges of $G_i$, where $\Delta_i$ is the maximum degree of $G_i$. This means that each machine needs to send at most $\tilde{O}(|E(G_i)|/k^2 + \Delta_i/k)$ messages over a specific communication link with high probability. In other words, the $i^{th}$ round of $A_C$ can be simulated in $\tilde{O}(|E(G_i)|/k^2 + \Delta_i/k)$ rounds, and, by taking a union bound, the same is true for all rounds in $[1, T_C(n)]$. By summing up over all rounds of $A_C$, we can conclude that the number rounds needed to simulate $A_C$ is

$$\tilde{O}\left(\sum_{i=1}^{T_C(n)} \left(\frac{|E(G_i)|}{k^2} + \frac{\Delta_i}{k}\right)\right) = \tilde{O}\left(\frac{M}{k^2} + T_C(n)\left\lceil \frac{\Delta'}{k}\right\rceil\right),$$

where the equality is because of the following facts: (1) $\sum_{i=1}^{T_C(n)} |E(G_i)| = O(M)$ since $|E(G_i)|$ is at most two times the number of messages sent by all nodes in the $i^{th}$ round, and (2) $\Delta_i \leq \Delta'$. This proves (a).

Proof of (b): We first slightly modify the previous simulation to simulate broadcast algorithms: Note that if $A_C$ is a broadcast algorithm, then for the $i^{th}$ round ($i \geq 1$) of algorithm $A_C$, if a node $u$ belonging to machine $p_1$ sends messages to nodes $v_1, \ldots, v_j$ ($j \geq 1$) belonging to machine $p_2$, we know that $u$ sends the same message to $v_1, \ldots, v_j$. Thus, when we simulate this round $A_C$, we will let machine $p_1$ send only one message to $p_2$, instead of $j$ messages. Then, machine $p_2$ will pretend that this message is sent from $u_1$ to all nodes belonging to $p_2$ that have an edge to node $u$. (We cannot specify the destination nodes $v_1, \ldots, v_j$ in this message as this might increase the length of the message significantly.)
CHAPTER 9. K-MACHINE MODEL AND ALGORITHMS

We now analyze this new simulation. We show that this simulation finishes in \( \tilde{O}(R/k + T(n)) \) rounds. Let \( R_i \) be the number of nodes that perform a broadcast in round \( i \) of the run of \( A_C \) in the clique model, and note that \( R = \sum_{i=1}^{T(n)} R_i \). According to Lemma 9.1(a), the number of nodes contributing to \( R_i \) broadcasts that are assigned to a single machine is \( \tilde{O}(\lfloor R_i/k \rfloor) \) w.h.p.; in other words, w.h.p., each machine contains \( \ell_i = \tilde{O}(\lfloor R_i/k \rfloor) \) of the \( R_i \) nodes. Thus, for every \( i \), we instruct algorithm \( A \) to simulate these \( R_i \) broadcasts in the \( k \)-machine model in \( \lfloor \ell_i \rfloor \) rounds. Since \( A_C \) takes at most \( T_C(n) \) rounds, we can take a union bound, and it follows that algorithm \( A \) takes \( \tilde{O}(R/k + T(n)) \) rounds in the \( k \)-machine model.

9.4 Algorithms for some problems

We now consider various important graph problems in the \( k \)-machine model. Observe that the simple solution of aggregating the entire information about the input graph \( G \) at a single machine takes \( O(m/k) \) rounds; thus we are only interested in algorithms that beat this trivial upper bound.

9.4.1 Breadth-First Search Tree (BFS)

To get an intuition for the different bounds obtained by applying either Theorem 9.1(a) or Theorem 9.1(b) to an algorithm in the clique model, consider the problem of computing a breadth-first search (BFS) tree rooted at a fixed source node. If we use Theorem 9.1(a) we get a bound of \( \tilde{O}(m/k^2 + D\lfloor \Delta/k \rfloor) \) rounds. In contrast, recalling that each node performs \( O(1) \) broadcasts, Theorem 9.1(b) yields \( \mathcal{P}^{k}(\text{BFS}) \in \tilde{O}(n/k + D) \). We will leverage these bounds when considering graph connectivity and spanning tree verification below.

9.4.2 Minimum Spanning Tree (MST), Spanning Tree Verification (ST) and Graph Connectivity (Conn).

We use the GHS algorithm for computing the MST of an input graph when the graph is a clique (Exercise 7.2). Recall that the GHS algorithm proceeds by merging “MST-fragments” in parallel; initially each vertex forms a fragment by itself. In each of the \( O(\log n) \) phases, each fragment computes the minimum outgoing edge (pointing to another fragment) and tries to merge with the respective fragment. Computing the minimum outgoing edge as well as merging can be done by broadcasting. In each phase all nodes do broadcasting. This
yields a total broadcast complexity of $O(n \log n) = \tilde{O}(n)$ and thus Theorem 9.1(b) readily implies the bound of $\tilde{O}(n/k)$.

We can use an MST algorithm for verifying graph connectivity which in turn can be used for ST. We assign weight 1 to all edges of the input graph $G$ and then add an edge with infinite weight between any pair of nodes $u, v$ where $(u, v) \notin E(G)$, yielding a modified graph $G'$. Clearly, $G$ is disconnected iff an MST of $G'$ contains an edge with infinite weight. This yields the same bound that we have for MST, namely $\tilde{O}(n/k)$.

We now describe how to verify whether an edge set $S$ is an ST, by employing a given algorithm $A$ for Conn. Note that, for ST verification, each machine $p$ initially knows the assumed status of the edges incident to its nodes wrt. being part of the ST, and eventually $p$ has to output either YES or NO. First, we run $A$ on the graph induced by $S$ and then we compute the size of $S$ as follows: Each machine locally adds 1 to its count for each edge $(u, v) \in S$, if $p$ is the home machine for vertices $u, v$. Otherwise, if one of $u$ or $v$ reside on a different machine, then $p$ adds $1/2$. Then, all machines exchange their counts via broadcast, which takes 1 round (since each count is at most $n$ and $W \in \Theta(\log n)$) and determine the final count by summing up over all received counts including their own. Each machine outputs YES iff (1) the output of the Conn algorithm $A$ returned YES and (2) the final count is $n - 1$. Thus we get the same bounds for ST verification as for graph connectivity.

Recalling that we can compute a BFS in $\tilde{O}(m/k^2 + D[\Delta/k])$ rounds, it is straightforward to see that the same bound holds for Conn (and thus also ST verification): First, we run a leader election algorithm among the $k$ machines. This can be done in $O(1)$ rounds and using $\tilde{O}(\sqrt{k})$ messages using Algorithm 7. The designated leader machine then chooses an arbitrary node $s$ as the source node and executes a BFS algorithm. Once this algorithm has terminated, each machine locally computes the number of its vertices that are part of the BFS and then computes the total number of vertices in the BFS by exchanging its count (similarly to the ST verification above). The input graph is connected iff the BFS contains all vertices.

### 9.4.3 PageRank

The PageRank problem is to compute the PageRank distribution of a given graph (may be directed or undirected). A distributed page rank algorithm based on the distributed random walk algorithm is as follows: Initially, each node generates $\Theta(\log n)$ random walk tokens. A node forwards each token with probability $1 - \delta$ and terminates the token with probability $\delta$ (called the reset probability). Clearly, every token will take at most $O(\log n/\delta)$ steps with
high probability before being terminated. We can show that these steps can be implemented in $O(\log^2 n)$ rounds in the clique model. Since this requires $O(n \log^2 n / \delta)$ messages to be sent in total, Theorem 9.1(a) yields that, for any $\delta > 0$, there is a randomized algorithm for computing PageRank in the $k$-machine model such that $\mathcal{R}_1^k(\text{PageRank}) \in \tilde{O}(\frac{n}{\delta^2})$.

9.5 Exercises

Exercise 9.1. You are given $n$ numbers and the goal is to sort them using a distributed algorithm in the $k$-machine model. The $n$ numbers are partitioned across the $k$-machines ($n > k$) in the following way: each machine gets an (arbitrary) subset of $n/k$ numbers (assume $k$ divides $n$). Give an $\tilde{O}(n/k^2)$-round (note that $\tilde{O}$ notation can hide polylog($n$) terms) distributed algorithm in the $k$-machine model for sorting the $n$ numbers; at the end, for every number, some machine should know the sorted index of that number (e.g., it may be the machine that initially contains that number, but can be any other machine as well). Note that $O(n/k)$ round algorithm is trivial: simply send all the $n$ numbers to one machine and that machine will sort all the numbers locally.

Hint: One can use randomized quicksort. Recall that in randomized quicksort, you choose a random element in the set as pivot, partition the elements into two parts based on the pivot (one part containing all elements smaller than the pivot and the other larger), and then recursively sort the two parts.

Exercise 9.2. Give an efficient algorithm to solve the single-source shortest path problem in the $k$-machine model. You can assume that you are given an undirected weighted graph $G$ with a source node $s$ and the goal is to find the shortest path distance from every node to $s$ (just finding the distance is enough, the path itself is not needed). Show correctness of your algorithm and analyze the round complexity in the $k$-machine model.
Chapter 10

Basics of Asynchronous Systems

So far, we have assumed that every message sent in a round is delivered by the same round and that all nodes compute in lock-step synchrony. While the synchronous model is a convenient abstraction that lets us study distributed algorithms in a round by round fashion, many real networks such as the internet are inherently asynchronous: instead of being delivered instantaneously, a message might be delayed for some arbitrary amount of time and nodes might perform their computing steps at vastly different speeds.

In an asynchronous system nodes take steps according to some given schedule that is a priori unknown to the algorithm. If such a step $s$ is triggered by the receipt of a message, we say that $s$ is a receive step. On the other hand, we say that a step is a send step if node $u$ sends out a set of messages. For convenience, we assume that a node can perform local computation in both, send and receive steps. From the above description, it is clear that each message is sent in exactly one send step and received in exactly one receive step. Figure 10.1 shows a graphical representation of an asynchronous algorithm. The space-time diagram simply lists the steps that each node takes over the passage of time (from left to right) where send steps and receive steps of messages are connected by a directed edge.

10.1 Measuring Time in an Asynchronous System

Recall that a node can very easily measure time in the synchronous model by simply counting the number of rounds. This means that when a node starts round

---

1To fully capture the harsh reality of distributed networks in the real world, we would also need to account for messages being lost and the failure of nodes. We defer the treatment of fault-tolerance to a later chapter.
In an asynchronous system, however, nodes take atomic steps at different speeds, thus a specific number of steps at one node does not necessarily imply anything on the progress of the computation at other nodes. This means that we cannot simply count the steps that a node takes to obtain an adequate measure of time. Nevertheless, many (asynchronous) distributed algorithms require some amount of coordination between nodes, which in turn is greatly facilitated if nodes can measure each other’s progress.

### 10.1.1 Logical Time and the Happens Before Relation

We start our treatment by classifying how computing steps can be related. We say that step $s_i$ *happens before* step $s_j$ if one of the following is true:

1. both of them happen at the same node and $s_i$ precedes $s_j$,
2. $s_i$ is the send step of a message $M$ and $s_j$ is the receive step of $M$, or
3. there is some step $s_k$ such that $s_i$ happens before $s_k$ and $s_k$ happens before $s_j$.

We can match the graphical representation of the space-time diagram to the happens before relation by considering the space-time graph (see Figure 10.1 (right)) where we add directed edges between subsequent steps at the same node. This also gives us a more intuitive (but equivalent) way to define that “$s_i$ happens before $s_j$” if and only if there is a path in the directed space-time graph from $s_i$ to $s_j$. For example, in the run depicted by Figure 10.1 we can see that step $s_3$ happens before step $s_8$ as there is a path $s_3 \rightarrow s_5 \rightarrow s_7 \rightarrow s_8$ in the space-time graph. This motivates us to use the notation $s_i \rightarrow s_j$ as a shorthand for saying that $s_i$ happens before $s_j$, and call a directed path in the space-time graph a causal chain.
10.1. MEASURING TIME IN AN ASYNCHRONOUS SYSTEM

10.1.1 Logical Clocks

We now describe how nodes can locally observe the happens before relation by implementing logical clocks. To this end, each node \( u \) keeps a local integer counter \( L_u \) that is initialized to 0. Whenever \( u \) sends a message, it piggybacks the current value of \( L_u \) on top of this message. There are 2 ways that \( u \) updates \( L_u \): If \( u \) performs a receive step and receives a message \( M \) that contains the counter value \( L_v \) of some node \( v \), it sets \( L_u := \max(L_u, L_v) + 1 \). In a send step, on the other hand, \( u \) simply increases \( L_u \) by 1. See Figure 10.2 for an example of the above algorithm. The following lemma is immediate from our discussion:

**Lemma 10.1 (Logical Time).** Suppose that step \( s_i \) happens before step \( s_j \). Then the logical clock value at \( s_i \) is smaller than the clock value at \( s_j \).

As we have seen above, logical time provides nodes a simple way to measure each others progress. Logical clocks, however, have the shortcoming that a step \( s \) that has clock value \( i \) does not necessarily “happen before” a step \( s' \) with a larger clock value \( j \) if \( s \) and \( s' \) occur at different nodes. For example, in Figure 10.2, the first step at node \( u \) has a logical clock value of 2 and the first step at node \( w \) has a clock value of 1, but neither step precedes the other according to our notion of “happens before”. The problem lies in the 1-dimensionality of logical time: we are trying to use a single integer value to represent the progress of up to \( n \) independent nodes. That is, we are losing some information by mapping a partial order (the happens before relation) into a linear order (logical time).

10.1.2 Vector Clocks

To avoid this pitfall, we will implement vector clocks instead: Each node \( u \) updates a \( n \)-dimensional vector \( \vec{V}_u \) of positive integers that is initialized to the 0-vector and is indexed by the node ids. Node \( u \)'s vector \( \vec{V}_u \) stores a value for each node \( w \), denoted by \( \vec{V}_u[w] \).
We now describe the rules for updating vector clocks in detail: Similarly as in the case of logical clocks, each node $u$ attaches its current value of $\vec{V}_u$ to each message that it sends. Whenever $u$ takes a step, it increases $\vec{V}_u[u]$ by 1. In addition, when receiving a message sent by a node $w$, node $u$ sets $\vec{V}_u[v] = \max(\vec{V}_u[v], \vec{V}_w[v])$, for every $v \neq u$. See Figure 10.3 for an example of how vector clocks progress over time.

**Lemma 10.2.** Consider a step $s$ at node $u$ and suppose that $\vec{V}_u(s)[v] = \lambda > 0$, for some $v$. Then, the first $\lambda$ steps $s_1, \ldots, s_\lambda$ of node $v$ happen before step $s$, i.e., there exists a causal chain $s_1 \rightarrow s_\lambda \rightarrow s$.

**Proof.** The lemma is trivially true if $u = v$ thus we focus on the case where $u$ and $v$ are distinct. How does node $u$ update its entry $V_v(s)[v]$? Recall that each node advances its own entry (i.e. $V_v(u)$) in each step and updates entries associated with other nodes by using the maximum rule upon receiving a message. Hence there must be a step $s_u$ where node $u$ receives a message with a vector clock value of $\vec{V}_w_1[v] = \lambda$, sent by some node $w_1$ in step $s_{w_1}$. We consider 2 cases:

1. $v \neq w$: Here we can apply the same argument to $w_1$ that we have applied to $u$ above. Namely, there must be an earliest step $s'_{w_1}$, where node $w_1$ has received a message with an attached vector clock value of $\vec{V}_{w_2}[v] = \lambda$, which was sent by some node $w_2$ in some step $s_{w_2}$, and also $s_{w_2} \rightarrow s'_{w_1} \rightarrow s_{w_1} \rightarrow s_u \rightarrow s$. If $w_2 = v$ we proceed to case 2. Otherwise we repeat the same argument for $w_2$ and construct the causal chain $s_{w_3} \rightarrow s'_{w_2} \rightarrow s_{w_2} \rightarrow s'_{w_1} \rightarrow s_{w_1} \rightarrow s_u \rightarrow s$ and so forth, yielding a sequence of steps at nodes $w_1, w_2, \ldots$. At each node $w_i$ ($i \geq 1$), we focus on the earliest event where $w_i$ has set its vector clock entry of $v$ to $\lambda$ by using the maximum rule. This ensures that, after repeating this argument at most $n - 1$, we reach case 2.

2. $v = w$: Note that $v$ must have increased $V_v[v]$ exactly $\lambda$ times by taking $\lambda$ steps. By the definition of the happens before relation, each one of these steps happens before step $s_w$, and since $s_w \rightarrow s$ we are done.

We will now show a one-to-one correspondence between vector clocks and the happens before relation. Recall that, when comparing two vectors $\vec{V}_1$ and $\vec{V}_2$, it holds that $\vec{V}_1 < \vec{V}_2$ if and only if (a) $\vec{V}_1[i] \leq \vec{V}_2[i]$, for every index $i$, and (b) there exists an index $j$ such that $\vec{V}_1[j] < \vec{V}_2[j]$.

Let $\vec{V}(s_1)$ and $\vec{V}(s_2)$ be the vector clock readings at step $s_1$ at node $u$, respectively step $s_2$ at $v$. It is easy to see that $\vec{V}(s_1) \neq \vec{V}(s_2)$ if $s_1 \neq s_2$. The following lemma motivates us to say that steps $s_1$ and $s_2$ are concurrent if and only if neither
10.1. MEASURING TIME IN AN ASYNCHRONOUS SYSTEM

Figure 10.3: A run of an asynchronous algorithm where each step is labeled with a vector clock reading.

\[ V(s_1) < V(s_2) \] nor \[ V(s_1) > V(s_2) \] hold, i.e., if the two vector clock values are incomparable.

**Theorem 10.1.** Let \( V(s_\alpha) \) be the vector clock reading of step \( s_\alpha \) at node \( u \) and let and \( V(s_\beta) \) be the vector clock reading of step \( s_\beta \) at \( v \), where \( s_\alpha \neq s_\beta \). Then \( s_\alpha \) happens before \( s_\beta \) if and only if \( V(s_\alpha) < V(s_\beta) \).

**Proof.** First, suppose that \( s_\alpha \) happens before \( s_\beta \). We will show that \( V(s_\alpha) < V(s_\beta) \). According to the definition of “happens before”, this means that there is a causal chain \( s_\alpha \rightarrow \cdots \rightarrow s_\beta \), starting at \( s_\alpha \) and ending at \( s_\beta \) such that, for any pair of subsequent steps \( s_k, s_{k+1} \) in this chain, exactly one of the following 2 statements is true:

(a) \( s_k \) and \( s_{k+1} \) happen at the same node \( w \);
(b) \( s_k \) is the send step of a message \( M \) and \( s_{k+1} \) is the receive step of \( M \).

In either case, the update rule of vector clocks ensures that \( V(s_k) < V(s_{k+1}) \) and since \( < \) is transitive, we have \( V(s_\alpha) < V(s_\beta) \).

Now, consider the case where \( V(s_\alpha) < V(s_\beta) \) and, for the sake of a contradiction, assume that \( s_\alpha \) does not happen before \( s_\beta \). Since the first part of our proof readily yields a contradiction to \( s_\beta \) happening before \( s_\alpha \), we only need to handle the case where \( s_\alpha \) and \( s_\beta \) are concurrent steps. Applying Lemma 10.2 to step \( s_\beta \) tells us that the first \( V_v(s_\beta)[u] \) steps of node \( u \) happen before step \( s_\beta \) at node \( v \). By assumption, it holds that \( V_v(s_\alpha)[u] \leq V_v(s_\beta)[u] \), and hence \( s_\alpha \) also happens before \( s_\beta \), as required.

□
10.1.3 Consistent Cuts

So far, we have seen that vector clocks sufficiently capture the causal relationships in an asynchronous system and allow a node to locally determine how many causally related steps have been performed by other nodes so far. We will now study the passage of time from the viewpoint of an external observer. Recall that, when talking about time in the synchronous model, we could simply point at the start of a specific round $r \geq 1$ to represent the vertical cut in the space-time diagram where each node has finished its computation regarding the previous $r - 1$ rounds. Similarly, if we can identify a point in time by looking at a vertical cut in the space-time diagram of the run of an asynchronous algorithm. For example, the vertical cut $C$ in Figure 10.4 (left), corresponds to the point in time when $u$ and $v$ each have taken 2 steps and $w$ has taken 1 step. Now let us consider the space-time diagram in Figure 10.4 (right). Here, the delivery of the first message from $v$ to $u$ has taken longer and thus step step $s_2$ of $u$ is shifted to the right. On the other hand, node $w$ receives the message from $u$ a bit sooner than and so step $s_6$ is drawn further on the left. It is important to keep in mind that nodes will perform the exact same computation in both space-time diagrams. This is because nodes cannot measure the passage of real-time between steps and hence observe the exact same environment in both scenarios. Therefore, the vertical cut $C$ on the left is equivalent to the cut $D$ on the right. Thus we can transform $C$ to $D$ and vice-versa using this rubber band transformation.

More formally, we can describe a cut as a vector of $n$-steps where the value of the $i$-th index represents the last step that the $i$-th node has performed before the cut. Thus we say that the set of the steps on the left side form the past of the cut and the set of steps on the right side are the future of the cut. Note that the cuts $C$ and $D$ of Figure 10.4 are equivalent according to this definition: either one of them represents a global snapshot of the asynchronous computation where $u$ has performed steps $s_2$ and $s_4$, node $v$ has taken $s_1$ and $s_5$, and node $w$ has taken step $s_3$. Since we can think of these cut through the space time diagram as global snapshot, the question arises if this is true for any cut. For example, consider the cut $E$ in Figure 10.4. Does $E$ represent some meaningful notion of global snapshot? In step $s_9$, node $w$ receives a message from node $v$ that was sent in step $s_7$. But according to the cut $E$, step $s_9$ lies in the past whereas step $s_7$ is in the future, contradicting the fact that messages cannot travel backwards in time. In other words, the cuts $C$ and $D$ are both “consistent” with our intuitive notion of snapshot whereas $E$ is inconsistent. Fortunately, the happens-before relation provides us with an intuitive criterion to identify whether a cut is consistent. To this end, we define the causal past of a step $s_j$ to be the set of all steps $s_j$ such that $s_j$ happens before $s_i$. We now formally define a cut $C = (s_1, \ldots, s_n)$ to be
10.1. MEASURING TIME IN AN ASYNCHRONOUS SYSTEM

consistent, if and only if, for each \( s_i \), the causal past of \( s_i \) is in the past of \( C \).

**Observation 10.1.** In the space-time diagram, a cut is consistent, if and only if, there is a rubber band transformation of the time-axis that does not violate the happens before relation.

Note that Observation 10.1 excludes messages that travel backwards in time such as in cut \( E \) of Figure 10.4.
Appendices
Appendix A

Asymptotic Notation

Algorithmic analysis will not be as elegant as it is if not due to the fact that we emphasise on asymptotic and approximate analysis, rather than exact analysis for a given input size. In other words, we mainly care about how the algorithm's performance scales with the problem size in a reasonably approximate manner that is approximate up to a constant factor (i.e., a constant that is independent of the problem size). This allows the algorithm analyzer to focus on quantifying the dominant function that determines the algorithm's performance, while avoiding cumbersome details.

Consider two positive functions \( g(n) \) and \( f(n) \) on natural numbers. Think of \( n \) as the input size of the algorithm; and \( f \) (and \( g \)) as denoting the runtime (of an algorithm) as a function of the input size.

A.1 Big-O notation

A function \( g(n) \) is “Big-O” of \( f(n) \), i.e., \( g(n) = O(f(n)) \), if there is some positive constant \( c > 0 \) (independent of \( n \)), such that \( cf(n) \) grows asymptotically (i.e., for large \( n \)) at least as “fast” as \( g(n) \). (See Figure A.1) In other words, \( g(n) \leq cf(n) \), for large \( n \), i.e., for all \( n \geq n_0 \) (\( n_0 \) is some fixed number).

Informally, we will say that if \( g(n) = O(f(n)) \), then \( f(n) \) (possibly, multiplied with a large-enough constant factor) “dominates” or “upper bounds” \( g(n) \). Formally we have the following definition.

**Definition A.1.1.** Big-O] Given two (positive) functions \( f(n) \) and \( g(n) \), we say \( g(n) = O(f(n)) \) if there is a positive constant \( c \) (independent of \( n \)), such that for any \( n \geq n_0 \), \( g(n) \leq cf(n) \).

The above definition is usually cumbersome to use for showing how two functions are related. An alternate, but easier way, is to define Big-O using limits as follows:
Figure A.1: Illustration of the Big-O notation. In both figures, \( g(n) = O(f(n)) \). Left: \( f(n) \geq g(n) \) for all \( n \geq n_0 \), where \( n_0 \) is a fixed constant. Note that what happens when \( n < n_0 \) is irrelevant. Right: \( g(n) \geq f(n) \), but \( g(n) \leq 5f(n) \), for all \( n \).

**Definition A.1.2.** Given two (positive) functions \( f(n) \) and \( g(n) \), we say \( g(n) = O(f(n)) \) if

\[
\lim_{n \to \infty} \frac{g(n)}{f(n)} \leq c.
\]

where \( c > 0 \) is a fixed constant (assuming the limit exists). In other words, the limit is bounded above by a fixed constant.

The above definition assumes that the limit exists.\(^1\)

### A.2 Big-Ω notation

Big-Ω notation can be considered a complement of the Big-O notation. If \( f(n) = O(g(n)) \) then we can say that \( g(n) = \Omega(f(n)) \). Informally, we will say that if \( g(n) = \Omega(f(n)) \), then \( f(n) \) (possibly, multiplied by a small-enough constant factor) is “dominated by” or “lower bounds” \( g(n) \).

Formally, we can define Big-Ω as we did for Big-O.

**Definition A.2.1.** A (positive) function \( g(n) \) is “Big-Ω” of (another positive) \( f(n) \), i.e., \( g(n) = \Omega(f(n)) \), if there is a positive constant \( c > 0 \) (independent of \( n \)), such that for any \( n \geq n_0 \), \( g(n) \geq cf(n) \).

Alternatively, using limits we have the definition:

\(^1\)Otherwise we can modify the definition to take the lim sup, e.g., see Example A.1. However, in most of the functions that we will encounter, limits exist and we don’t have to worry about this technicality.
**A.3. Big-Theta (Θ) Notation**

**Definition A.2.2.** A (positive) function $g(n)$ is “Big-$\Omega$” of (another positive) $f(n)$ if

$$\lim_{n \to \infty} \frac{g(n)}{f(n)} \geq c,$$

where $c > 0$ is a fixed positive constant (assuming the limit exists). In other words, the limit is bounded below by a fixed constant.

The above definition assumes that the limit exists (otherwise we can modify the definition to take the lim inf).

**A.3 Big-Theta (Θ) Notation**

Big-Theta notation is used to compare functions that are both Big-O and Big-$\Omega$ of each other, i.e., in other words have the same asymptotic behavior. More precisely, two positive functions $g(n)$ and $f(n)$ are said to be “Big-Theta” of each other if their growth is asymptotically within a constant factor of each other. We can formally define it as follows.

**Definition A.3.1.** A (positive) function $g(n)$ is “Big-Theta” of (another positive) $f(n)$ if there are some fixed positive constants $c_1$ and $c_2$ such that, $c_1 f(n) \leq g(n) \leq c_2 f(n)$, for all $n > n_0$. Equivalently, $g(n) = \Theta(f(n))$ if $g(n) = O(f(n))$ AND $g(n) = \Omega(f(n))$.

Informally, we will say that if $g(n) = \Theta(f(n))$, then both $f(n)$ and $g(n)$ are of the “same order”, i.e., they are essentially the same function (as far as asymptotic growth is concerned).

Alternatively, using limits we have the definition:

**Definition A.3.2.** A (positive) function $g(n)$ is “Big-Theta” of (another positive) $f(n)$ if

$$\lim_{n \to \infty} \frac{g(n)}{f(n)} = c,$$

where $c > 0$ is a fixed positive constant (assuming the limit exists).

**A.4 Little-o and Little-ω notations**

Sometimes, we would like asymptotic notation to capture a function that grows strictly slower than some other function. For this we use the Little-o notation which can be considered as a stronger form of the Big-O notation. If $f(n)$ is $o(g(n))$ then $f(n) = O(g(n))$, but not vice-versa.

**Definition A.4.1.** Given two functions, $g(n)$ and $f(n)$, we say that $g(n) = o(f(n))$ if for any positive constant $c$ there is a constant $n_c$ such that for any $n > n_c$, $g(n) < cf(n)$.
The above definition says that if \( g(n) \) is \( o(f(n)) \), i.e., \( g(n) \) grows strictly slower than \( f(n) \) or \( f(n) \) grows strictly faster than \( g(n) \), then no matter what constant \( c \) you choose (however small it may be), when \( n \) becomes large enough (i.e., \( n > n_c \), where \( n_c \) is a constant that depends on \( c \)), \( g(n) \) is dominated by \( cf(n) \). In other words, \( cf(n) \) always starts dominating \( g(n) \), however small \( c \) is, after a certain point (when \( n \) becomes large).

Alternatively, using limits we have the definition:

**Definition A.4.2.** A (positive) function \( g(n) \) is “Little-o” of (another positive) \( f(n) \) if

\[
\lim_{n \to \infty} \frac{g(n)}{f(n)} = 0.
\]

Similarly, we would like asymptotic notation to capture a function that grows strictly faster than some other function. This is the Little-\( \omega \) notation which can be considered as a complement of the Little-o notation. That is if \( f(n) = o(g(n)) \), then \( g(n) = \omega(f(n)) \).

**Definition A.4.3.** Given two functions, \( g(n) \) and \( f(n) \), we say that \( g(n) = \omega(f(n)) \) if for any positive constant \( c \) there is a constant \( n_c \) such that for any \( n > n_c \), \( g(n) > cf(n) \).

Alternatively, the above can be expressed using limits as:

**Definition A.4.4.** A (positive) function \( g(n) \) is “Little-\( \omega \)” of (another positive) \( f(n) \) if

\[
\lim_{n \to \infty} \frac{g(n)}{f(n)} = \infty.
\]

(assuming the limit exists)

### A.5 Examples

We next illustrate the above notations with some examples.

**Example A.1.**  
1. \( 2n = O(n) = \Theta(n) \). *Constants are ignored! Big-O, Big-Omega, and Big-Theta notation “hides” constant factors.*
2. \( n^3 + 2^{1000}n^2 + 2^{100000}n = O(n^3) = \Theta(n^3) \).
3. In general for any polynomial

\[
p(n) = \sum_{i=0}^{d} a_in^i
\]

where \( a_i \) are constants and \( a_d > 0 \), we have \( p(n) = \Theta(n^d) \).
4. \( n = O(n^2) \), but \( n \neq \Theta(n^2) \). That is, \( n^2 \) grows “strictly” faster than \( n \).
5. \( \lg n = O(n) \) (where \( \lg n = \log_2 n \)).
6. \( n^{1000} = O(2^n) \), but \( n^{1000} \neq \Theta(2^n) \).

7. \(1/n = o(1)\)

8. \( n^{1000} = o(1.1^n) \)

9. \( n^2 = \omega(n) \)

10. \( n = (\cdot)n^{1 + \sin n} \)

\[
\limsup_{n \to \infty} \frac{n}{n^{1 + \sin n}} = \infty
\]
\[
\liminf_{n \to \infty} \frac{n}{n^{1 + \sin n}} = 0
\]

Hence the two functions are not asymptotically comparable via little-o or little-\( \omega \) notation.
Appendix B

Mathematical Induction

Mathematical induction is a simple but very powerful technique to prove mathematical theorems. It is also extremely useful in proving correctness of algorithms as well as in understanding the insight behind their design; it is also useful in analyzing the performance of algorithms. Before we illustrate mathematical induction with some examples, a few words about the general technique. Suppose we want to prove some mathematical statement that depends on some parameter $n$ which can be any natural number. Examples are: (1) the sum of the first $n$ numbers is $n(n+1)/2$; (2) $2^n \geq 2n$; (3) the running time of an algorithm on input size $n$ is at most $2n$ etc. The goal is to show that the mathematical statement is true, i.e., it is true for all natural numbers. Note that it is generally easy, at least in principle, to verify the statement for a particular value of $n$, e.g., if $n = 100$, the first statement can be checked by simply summing up the first 100 numbers (may take a while) and verifying that it equals $(100)(101)/2$. However, this is not a valid approach to proving the statement for every natural number. Mathematical induction is a technique for precisely doing this. (Note that in many cases, there are alternative ways to prove such statements, but induction is the most straightforward and “automatic” way to do that; indeed there are important mathematical theorems where the only known proof uses induction.)

B.1 The General Technique

How does mathematical induction work? Suppose we want to show that a mathematical statement is true for all natural numbers $n \geq 0$. Mathematical induction consists of two steps: (1) Base step and (2) Induction Step. The base step is verifying whether the statement is true for $n = 0$. This is, of course, easy, since we have to verify for just one value. The induction step is the crucial step: assuming that the statement is true for natural number $k$, we have to prove that the statement is true for $k + 1$. It is important to
note that $k$ is a variable and not any fixed number, and thus proving the induction step involves proving the statement for any (general) $k$. The assumption that the statement is true for natural number $k$ is usually called the induction hypothesis. If we show these two steps — the base step and the induction step — then by mathematical induction it follows that the statement is true for all natural numbers. Why is this the case? This is not difficult to see if we iterate through the numbers. The base step shows that the statement is true for $n = 0$. Applying the induction step for $k = 0$ (since we show it for any number $k$), it holds for $k = 1$. Since it holds for $k = 1$, we can again use the induction step to show the statement for $k = 2$. Again, since it holds for $k = 2$, we can again use the induction step to show the statement for $k = 3$ and so on. Repeating this argument, it is clear that the statement holds for every natural number. This is the essence of a proof by mathematical induction.

The power of mathematical induction — which naturally underlies recursive algorithms (and iterative algorithms as well) — comes from the induction hypothesis which simply assumes that the statement is true for $k$ (somehow by “magic”!). This gives us a lot of “ammunition” to prove that the statement is true for $k + 1$.

### B.1.1 Sum of first $n$ natural numbers

Let’s illustrate the technique of mathematical induction by proving a basic mathematical statement (which also arises frequently in analysis of algorithms): the sum of the first $n$ natural numbers is $n(n + 1)/2$.

**Theorem B.1.**

\[ 0 + 1 + 2 + \cdots + n = \sum_{i=0}^{n} i = \frac{n(n + 1)}{2}. \]

**Proof.** We prove by using mathematical induction on the variable $n$. (Note that it is a good idea to say the variable that we are applying induction to. In this example, it is obvious since there is only one variable, but in others, there may be more than one variable in the statement, and it better to be explicit about it. We will see more examples throughout this book.)

We have to show the following two steps:

- **Base step: $n = 0$:** The statement is trivially true, since $0(1)/2 = 0$ which is indeed the sum of the first number (0).
- **Induction step:** Assume that the statement is true for $k$ (induction hypothesis). We will prove that the statement is true for $k + 1$.

By the induction hypothesis, we have

\[ 0 + 1 + 2 + \cdots + k = \frac{k(k + 1)}{2}. \]
Using the above, we have to prove that the following statement holds:

\[ 0 + 1 + 2 + \cdots + k + k + 1 = \frac{(k + 1)(k + 2)}{2} \]  \hspace{1cm} (B.1)

How can we show that? We use the induction hypothesis that already tells us the value of the sum up to \( k \). Thus the left hand side of Equation (B.1) becomes

\[ \frac{k(k + 1)}{2} + k + 1 \]

which simplifies to \( \frac{(k+1)(k+2)}{2} \) which is the right hand side of Equation (B.1) as desired.

Thus we have proved the induction step and hence the statement is proved by mathematical induction. \( \square \)

**B.1.2 Coloring regions formed by lines**

We will prove a simple, but somewhat surprising result on coloring regions formed by lines. Consider \( n \) distinct (infinite) lines drawn in a plane. A region is an area of the plane that is enclosed by lines; note that a region can be finite (closed) or infinite (open). The goal is to assign colors to regions so that no two adjacent regions (i.e., regions that share a line as a boundary) have the same color — such a coloring is called **valid**. For example, if there is only one line, it partitions the plane into two (adjacent) regions and if there are two intersecting lines then it partitions the plane into four regions where each region is adjacent to two other regions. How many colors are needed to validly color regions formed by \( n \) lines? The following theorem shows that only two colors are needed.

**Theorem B.2.** Two colors are needed to validly color all regions formed by any number of lines.

**Proof.** The proof is by induction on \( n \), the number of lines.

**Base step:** \( n = 1 \). Then the line partitions the plane into two regions; color one region by blue and the other by red. Clearly this coloring is valid.

**Induction step:** Assume that the statement is valid for \( k \) lines. We will prove the statement when there are \( k + 1 \) lines. Consider \( k + 1 \) distinct lines drawn in a plane. We will demonstrate the existence of a valid coloring of the regions formed by these \( k + 1 \) lines using two colors. Consider the set of \( k + 1 \) lines and remove one line from the set. This leaves us with \( k \) lines and by the induction hypothesis, there is a valid coloring of the regions formed by it using 2 colors — red and blue. Assume this valid coloring. Now let’s add the \( (k+1) \)th line and we will show how to modify the colors of the regions to still have a valid coloring. We leave the colors of all regions that are on one side of the \( (k+1) \)th line unchanged and reverse the colors of the regions on the other side. We
claim that this is a valid coloring. Consider the regions formed by \( k \) lines. When the 
\((k + 1)\)th line was added, it cuts some regions formed by the \( k \) lines into two parts —
one on each side. Since we reverse the color on one side of the \((k + 1)\)th line, these “cut”
regions have a valid coloring. Now consider the regions that are not cut by the \((k + 1)\)th
line, i.e., those that are entirely on either side of the line. These are (already) validly
colored by the induction hypothesis, since these are regions formed by \( k \) lines. 

**Algorithm via Induction Proof.**

The above induction proof also suggests a natural algorithm that gives a valid
coloring of the regions formed by \( n \) lines. The algorithm is naturally incremental. Start
with one line and use two colors to color the two regions on either side — this is the
base case. Then add one line at a time. After adding a line, simply reverse the colors of
the regions on one side of the added line, leaving the other side unchanged — this is
the strategy used in the induction step. Clearly this algorithm gives a valid coloring and
the proof of its correctness is the indeed the induction proof!

This example illustrates how algorithms can be designed by using induction proofs;
as an added bonus the proof of correctness of the algorithm follows from the induction
proof.

**Weak Induction.** The above form of mathematical induction is sometimes called the
“weak form of induction” as the induction hypothesis assumes only the validity of the
statement for value \( k \) (and not for values less than \( k \) — as done in the “strong form” as
explained in the next section). This is already useful in proving correctness of many
algorithms, especially iterative algorithms, which operate incrementally on the input
size. In particular, it is useful in proving “loop invariant” in a for or while loop. A loop
invariant is a property that holds in every iteration of the for loop; if it fails to hold
the loop terminates. Correctness of many algorithms with for and while loops involve
showing statements on loop invariants. Such examples are shown throughout the book.

**B.2 Strong Mathematical Induction**

In many cases, the induction hypothesis can be strengthened for “free.” Instead of
assuming that the statement is true for \( k \), we can assume that the statement is true
for all natural numbers \( n \leq k \) and then proceeding to show, under this assumption,
that the statement is true for \( k + 1 \). The strengthened hypothesis, called strong form of
induction, is extremely useful, leading to simpler proofs. It is especially useful in proving
correctness (and understanding) of recursive algorithms.
B.2.1 Fundamental Theorem of Arithmetic

We will illustrate the strong form of induction using the following example, which is an important theorem in mathematics: the fundamental theorem of arithmetic which states that every natural number greater than 1 can be expressed as a product of prime numbers. Recall that a prime number is a positive integer (greater than 1) that is divisible only by itself and by 1.

**Theorem B.3.** Every natural number $n > 1$ can be expressed as a product of prime numbers.

**Proof.** We will prove by induction on $n$.

Base case: $n = 2$ (note that this is the base case, as it is the smallest natural number for which the theorem is supposed to hold). Since 2 is prime, the statement is trivially true.

Induction step: Assume that the statement is true for all natural numbers from 2 to $k$ (induction hypothesis). Note that we are assuming the strong form of the induction which is needed for this proof. We will prove that the statement is true for $k + 1$.

There are two cases to consider. Suppose $k + 1$ is prime — in this case, the statement is trivially true. Suppose $k + 1$ is not a prime number. Then $k + 1$ can be decomposed into a product of two numbers $a$ and $b$, i.e., $k + 1 = ab$, where $1 < a < k + 1$ and $1 < b < k + 1$ (in fact they are both strictly smaller than $k$). Since $a$ and $b$ are strictly smaller than $k + 1$ and strictly greater than 1, induction hypothesis applies to both of them, and hence each of them can be written as a product of prime numbers. Hence $k + 1$ which is a product of $a$ and $b$ can be written as a product of prime numbers. \( \square \)

Note that the strong form was needed for the above proof, since $a$ and $b$ are numbers that are smaller than $k$.

B.3 Using Induction for Algorithm Analysis

One weakness of induction is that one needs to know the exact statement that one needs to prove beforehand. For example, we need to know that the sum of the first $n$ numbers is $n(n + 1)/2$ and then induction can be used to prove this as demonstrated in Theorem B.1. Induction does not seem to help to discover the fact that the sum of the first $n$ numbers is $n(n + 1)/2$; it merely serves to verify it. Can induction do better? In many cases, it can, as we show as follows. Suppose we want to solve a slightly easier problem that occurs very often in analysis of algorithms: we want a good bound (not the exact value) on some sum or, more generally, some function of $n$. To be concrete, suppose we want to show an upper bound on the sum of the first $n$ numbers (which actually arises in analysis of algorithms for sorting and selection) and assume that we
don’t know the exact formula. We want to show that the bound is $O(n^a)$ which usually suffices for algorithm analysis. We want a tight upper bound as possible, i.e., we want $a$ to be as small as possible. We know from Theorem B.1 that it is true for $a = 2$ (since $n(n+1)/2 \leq n^2$), but let’s use induction to find this.

Let’s set up the statement to be shown:

$$0 + 1 + 2 + \cdots + n \leq cn^a$$

where, $c$ and $a$ are fixed constants. We want to find the smallest $a$ for which the above statement is true ($c$ can be any fixed constant).

Base step: $n = 0$. The statement is clearly true for any fixed $a$ and $c$.

Induction step: Assume the statement is true for $k$. We will prove that it is true for $k + 1$ for a suitable $a$.

By induction hypothesis, $\sum_{i=0}^{k} i \leq ck^a$ and thus,

$$\sum_{i=0}^{k+1} i = \sum_{i=0}^{k} i + k + 1 \leq ck^a + k + 1.$$ 

For the induction step, we need to show that

$$ck^a + k + 1 \leq c(k+1)^a. \quad (B.2)$$

The above inequality will not be satisfied for $a = 1$, since that will imply that $k + 1 \leq c$ which is clearly not true (the left hand side increases with $k$, while the right hand side is a constant). Let’s try $a = 2$. Plugging $a = 2$ into the inequality (B.2) and simplifying yields $\frac{k+1}{k+2} \leq c$, which is true for all $k$ if $c = 1$. Hence if we take $c = 1$ and $a = 2$, the induction step is satisfied. Thus, we have shown, in fact, derived, that $\sum_{i=0}^{n} i \leq n^2$.

### B.4 Worked Exercises

**Worked Exercise B.1.** Show using mathematical induction that any integer whose sum of digits is divisible by 3 is divisible by 3 as well.

**Solution:**

Base Case: $n = 0$: Trivially true.

Induction hypothesis: Assume the statement is true for all integers less than $k$.

Induction step: We prove the statement that it is true for $k$.

Consider the number $k$ and let its representation in decimal form (digits) be $x_m x_{m-1} \cdots x_1 x_0$ — consisting of $m$ digits with $x_0$ being the least significant digit and $x_m$ being the most significant. In other words, $k = 10^m x_m + 10^{m-1} x_{m-1} + \cdots + 10 x_1 + x_0$. Let’s denote the sum of the digits of number $k$ to be $s_k = x_m + x_{m-1} + \cdots + x_1 + x_0$. Let’s assume that $s_k$ is
divisible by 3. Then we have to show that \( k \) is divisible by 3 as well.

Consider the integer \( k - 3 \). We have two cases:

Case 1: The last digit of \( k \), i.e., \( x_0 \) is greater than or equal to 3. Then the decimal representation of \( k - 3 \) is \( x_m x_{m-1} \ldots x_1 x_0 - 3 \) and the sum of its digits is \( s_{k-3} = x_m + x_{m-1} + \cdots + x_1 + x_0 - 3 = s_k - 3 \). Since \( s_k \) is divisible by 3 (by the above assumption), \( s_k - 3 \) is divisible by 3 as well. Hence the sum of the digits of number \( k - 3 \) is divisible by 3. By induction hypothesis (which applies to numbers less than \( k \)), \( k - 3 \) is divisible by 3. Hence \( k - 3 + 3 = k \) is divisible by 3 as well.

Case 2: The last digit of \( k \), i.e., \( x_0 \) is less than 3. Then the decimal representation of \( k - 3 \) can be computed as follows. The idea is to figure out the digits of \( k - 3 \). Let \( x_i \) (\( i > 0 \)) be the least significant non-zero digit, i.e., it is the first digit starting from the left (except, \( x_0 \)) that is non-zero.

(Example: if \( k = 83001 \), then \( x_i = 3 \), since that is the first non-zero digit from the left (apart from the leftmost digit). If we subtract 3 from 83001, then we get 82998, where \( x_i \) will be subtracted by 1, and then all 9s and then the last digit is gotten by subtracting 11 (10+1) from 3. This is because of the usual subtraction where we borrow 1 from \( x_i \) and then do it repeatedly till we reach \( x_0 \); that why have 9’s in between. Note that since the sum of the digits of 83001 is divisible by 3, the sum of the digits of 82998 is divisible by 3 as well. Next we generalize the above example.)

Then the sum of its digits of \( k - 3 \) is \( s_{k-3} = x_m + x_{m-1} + \cdots + x_i - 1 + 9 + \cdots + 9 + 10 + x_0 - 3 = x_m + x_{m-1} + \cdots + x_i + x_0 - 1 + 9 + \cdots + 9 + 10 - 3 = s_k + x_i - 1 + 9 + \cdots + 9 + 9 - 3 \) which is divisible by 3 since \( s_k \) is divisible by 3 and the sum \( 9 + \cdots + 9 + 9 - 3 \) is divisible by 3 (since each term is). The rest of the proof is now the same as Case 1.

Hence the sum of the digits of number \( k - 3 \) is divisible by 3. By induction hypothesis (which applies to numbers less than \( k \)), \( k - 3 \) is divisible by 3. Hence \( k - 3 + 3 = k \) is divisible by 3 as well.

B.5 Exercises

Exercise B.1. Prove that the sum of the squares of the first \( n \) natural numbers is \( \frac{n(n+1)(2n+1)}{6} \).

Exercise B.2. Using induction, give a tight bound on the following sum: \( \sum_{i=1}^{n} i^d \), where \( d \) is a fixed constant.

Exercise B.3. Prove by mathematical induction the following statement:

\[
\sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a - 1}
\]

where \( a \neq 1 \) is some fixed real number.
APPENDIX B. MATHEMATICAL INDUCTION

By the way, this is the sum of a geometric series, a useful formula that comes across often in algorithmic analysis.

Exercise B.4. Use mathematical induction to prove that $x^n - y^n$ is divisible by $x - y$ for all natural numbers $x, y$ ($x \neq y$), and $n$. (Hint: You will need strong induction.)

Exercise B.5. Show that for all natural numbers $n \geq 1$, $\binom{2n}{n} \geq 2^n$. Note that $\binom{2n}{n} = \frac{(2n)!}{n!n!}$.

Exercise B.6. Consider the set of first $2n$ positive integers, i.e., $A = \{1, 2, \ldots, 2n\}$. Take any subset $S$ of $n + 1$ distinct numbers from set $A$. Show that there two numbers in $S$ such that one divides the other.
Appendix C

Probability

We recall basic definitions and concepts of probability theory. We also present various probabilistic inequalities and bounds that are used throughout the book. These prove very useful in the analysis of randomized algorithms. We refer to [4][2] for more details on the concepts presented here.

C.1 Events, Probability, Probability Space

Consider an experiment with a finite (or countably infinite) number of outcomes. Each outcome is a simple event (or a sample point). The sample space is the set of all possible simple (elementary) events. An event is a union of simple events — a subset of the sample space.

Two events are said to be mutually exclusive if \( A \cap B = 0 \).

With each simple event \( s \) we associate a number \( Pr(s) \) which is called as the probability of \( s \).

**Definition C.1.1** (Probability Space). A probability distribution \( Pr \) on a sample space \( S \) is a mapping from events of \( S \) to real numbers such that it satisfies the following three axioms:

- \( Pr(A) \geq 0 \) for any event \( A \).
- \( Pr\{S\} = 1 \).
- For any (finite or countably infinite) sequence of pairwise mutually exclusive events \( A_1,A_2,\ldots \):

\[
Pr\{\cup_i A_i\} = \sum_i Pr\{A_i\}
\]

The pair \((S, Pr)\) is called a discrete probability space.
The probability of any event \( E \) can be computed as the sum of the probabilities of the simple events that it is composed of; this follows from the third axiom:

\[
Pr(E) = \sum_{s \in E} Pr(s).
\]

### C.2 Principle of Inclusion-Exclusion

The *inclusion-exclusion* principle gives a formula for computing the probability of the union of a set of events in terms of the probabilities of the individual events and their intersections.

**Statement C.1** (Inclusion-Exclusion Principle). Let \( E_1, E_2, \ldots, E_n \) be arbitrary events. Then

\[
Pr(\bigcup_{i=1}^n E_i) = \sum_i Pr(E_i) - \sum_{i<j} Pr(E_i \cap E_j) + \Pr_{i<j<k} Pr(E_i \cap E_j \cap E_k) - \ldots + (-1)^{i+1} \sum_{i_1 < i_2 < \ldots < i_l} Pr(\bigcap_{r=1}^l E_{i_r}) + \ldots.
\]

An important consequence of the inclusion-exclusion principle is a simple upper bound on the probability of the union of events known as **Boole’s inequality** or **union bound**.

**Statement C.2** (Boole’s inequality (union bound)). For any arbitrary sequence of events \( E_1, E_2, \ldots, E_n \):

\[
Pr(\bigcup_{i=1}^n E_i) \leq \sum_i Pr(E_i)
\]

The union bound is used often in upper bounding the occurrence of the union of a set of “bad” events. If this upper bound is small, then it implies that none of the bad events occur with high probability.

### C.3 Conditional Probability

**Definition C.3.1** (Conditional Probability). Given two events \( A \) and \( B \) conditional probability of \( A \) given \( B \) is defined as follows:

\[
Pr(A | B) = \frac{Pr(A \cap B)}{Pr(B)}
\]

In the above definition, by conditioning on \( B \) we restrict the sample space to the set \( B \). Thus the conditional probability can be considered as \( Pr(A \cap B) \) “normalized” by \( Pr(B) \).
Given two events $E_1$ and $E_2$, Bayes’ rule relates the conditional probability of the first given the second, to the conditional probability of the second given the first. This is useful to infer one conditional probability from the other.

**Statement C.3** (Bayes’ rule).

\[
Pr(E_1|E_2) = \frac{Pr(E_1 \cap E_2)}{Pr(E_2)} = \frac{Pr(E_2|E_1)Pr(E_1)}{Pr(E_2)}
\]

The following are some useful identities that involve computing the probability of the intersection of many events.

**Statement C.4.**

- $Pr(A \cap B) = Pr(A | B)Pr(B)$.
- $Pr(A \cap B \cap C) = Pr(A | B \cap C)Pr(B \cap C) = Pr(A | B \cap C)Pr(B | C)Pr(C)$.
- The following is a generalization of the above identity. Let $A_1, \ldots, A_n$ be a sequence of events. Let $E_i = \bigcap_{j=1}^{i} A_i$. Then
  \[
  Pr(E_n) = Pr(A_n | E_{n-1})Pr(E_{n-1}) = Pr(A_n | E_{n-1})Pr(A_{n-1} | E_{n-2})\ldots P(A_2 | E_1)Pr(A_1).
  \]

A fundamental concept that follows from conditional probability is that of independence.

**Definition C.3.2** (Independence of events). Two events $A$ and $B$ are said to be independent if

\[
Pr(A \cap B) = Pr(A) \times Pr(B),
\]

or (when $Pr(B) > 0$)

\[
Pr(A | B) = \frac{Pr(A \cap B)}{Pr(B)} = Pr(A).
\]

### C.4 The Birthday Paradox

We study a problem called the “Birthday Paradox”, that arises often in algorithm design and analysis. The problem also serves to illustrate basic concepts in probability theory.

Question: What is the probability that among $m$ people no two have the same birthday? We make the following two assumptions: (1) All birthdays are equally likely and (2) Birthdays are independent events.

To compute the above probability, we first define the sample space of the experiment. The sample space is the set of all vectors $S = \{(b_1, \ldots, b_m) | b_i \in [1, \ldots, N]\}$, where $b_i$ denotes the birthday of the $i$th person, and $N$ is the total number of different birthdays ($N = 365$ in Earth). We need to compute $Pr(E)$ where the event $E = \{(b_1, \ldots, b_m) | b_i \neq b_j$ for all $i \neq j\}$, i.e., the event $E$ is the set of all events where no two birthdays are the same.
How many different atomic events are counted in $E$? The number of possible $m$ different birthdays is $N(N - 1)(N - 2)\ldots(N - m + 1)$. Hence,

$$Pr(E) = \frac{N(N - 1)(N - 2)\ldots(N - m + 1)}{N^m}.$$ 

$$= \Pi_{i=0}^{m-1}(1 - i/N)$$

$$\leq \Pi_{i=0}^{m-1}e^{-i/N} = e^{-\sum_{i=0}^{m-1} i/N} = e^{-m(m-1)/2N}.$$ 

For $m = \sqrt{2N} + 1 \leq 28$, $Pr(E) < 1/e < 1/2$.

The apparent “paradox” in this problem is that significantly less number of people, i.e., only about $\sqrt{N}$ people (which is much smaller than $N$), are needed to have a good chance (about 50%) to have two people to have a common birthday.

**Alternate Analysis.** Assume that we choose one birthday after the other independently and uniformly at random from $[1\ldots N]$. Let the event $E_i$ denote: “the $i$th choice is different from the first $i - 1$ choices”. Then, we compute:

$$Pr(E) = Pr(\cap_{i=1}^m E_i) = Pr(E_1)Pr(E_2|E_1)Pr(E_3|E_2 \cap E_1)\ldots Pr(E_m|\cap_{i=1}^{m-1} E_i)$$

$$= 1(1 - 1/N)(1 - 2/N)\ldots(1 - (m - 1)/N)$$

$$= \Pi_{i=1}^m(1 - \frac{i - 1}{N}).$$

This gives the same result as before.

This analysis uses a very useful principle called the **Principle of Deferred Decisions.** In this principle, the idea is to defer the fixing of choices of events to “when they are needed” and not a priori (i.e., all at once). Typically, this type of analysis will use conditional probabilities. In the above analysis, to compute $Pr(E)$, we first computed $Pr(E_1)$, then $Pr(E_2|E_1)$ and so on. At each stage, we only fixed the event that determined the respective conditional probabilities, i.e., first $E_1$ which fixes $b_1$, and then the event $E_2$ conditioned on $E_1$, which fixes $b_2 \neq b_1$, and so on.

**C.5 Random Variable**

Random variable is a very useful concept in probability; it allows one to quantify the parameters associated with the events in a probability space as we see fit to model.

Let $(\mathcal{F}, Pr)$ be a discrete probability space. Let $V$ be a set of values. A random variable $X$ defined on $(\mathcal{F}, Pr)$ is a function

$$X : \mathcal{F} \to V.$$ 

The random variable $X$ allows one to “transfer” the probability function that is defined
on the set of events of \( \mathcal{F} \) to one that is defined on the set of values \( V \) in the following manner. Let \( \mathcal{E}(r) = \{ s \in \mathcal{F} \mid X(s) = r \} \) Then we can define:

\[
Pr(X = r) = Pr(\mathcal{E}(r)) = \sum_{s \in \mathcal{E}(r)} Pr(s).
\]

Two random variables \( X \) and \( Y \) (defined on the same sample space) are called independent if for all \( x \) and \( y \)

\[
Pr\{X = x \text{ and } Y = y\} = Pr\{X = x\} Pr\{Y = y\}
\]

### C.6 Expectation of a random variable

Random variable is a function (thus “variable” is actually a misnomer!), and thus it can take any value in its range. The probability function \( Pr \) gives the probability that each such value is taken, i.e., it gives the probability distribution over the values in \( V \). In many applications, we are less interested in the entire distribution of the probability over the values, but rather than a single (or few) parameter(s) that characterizes the distribution. A standard such parameter is the expectation, also called as the mean or the average of the random variable.

**Definition C.6.1** (Expectation of a random variable). The expectation of a discrete random variable \( X \) is

\[
E[X] = \sum_{i \in \text{range}(X)} i Pr(X = i).
\]

Thus, the expectation is a weighted sum over all possible values of the random variable, weighted by the corresponding probabilities of the values.

A very useful property that comes in handy in computing the expected value of a random variable is the linearity property of expectation stated as follows.

**Theorem C.1** (Linearity of Expectation). Given a sequence of random variables, \( X_1, X_2, \ldots, X_n \), we have

\[
E \left[ \sum_{i=1}^{n} X_i \right] = \sum_{i=1}^{n} E[X_i].
\]

The linearity property says that the expectation of the sum of random variables is equal to the sum of their individual expectations. Note that this is true, regardless whether the random variables are independent or not; in fact, it is the latter case that makes this property so useful in many applications. In many applications, this is commonly used to compute the expectation of a random variable (which is not directly easy to compute by applying the Definition C.6.1) by decomposing it into a sum of random variables whose expectations are easier to compute. A typical example is decomposing the random
variable into a set of indicator (also called Bernoulli or 0-1) random variables, which take only two values 0 and 1. The following example illustrates this paradigm.

**Problem C.1.** Assume that N customers checked their jackets in a restaurant. The jackets get mixed and while leaving the restaurant, each customer gets a random jacket, i.e., probability of getting any particular jacket is $1/N$. The goal is to compute the expected number of customers who get their own jacket.

**Solution:** It is difficult to compute $E[X] = \sum_{k=0}^{N} k \Pr(X = k)$. Instead, we define $N$ indicator random variables $X_i$, where $X_i = 1$ iff $i$ got her own jacket. That is, $X_i$ "indicates" whether customer $i$ gets her own jacket or not. It is much simpler to compute $E[X_i]$ for any $i$, since it takes only two values 0 and 1. Note that $X_i$s are not independent, but we can still apply linearity of expectation.

$$E[X_i] = 1 \cdot Pr(X_i = 1) + 0 \cdot Pr(X_i = 0) = Pr(X_i = 1) = \frac{1}{N}.$$  

$$E[X] = \sum_{i=1}^{N} E[X_i] = 1.$$  

**Problem C.2.** Let $a_1, a_2, \ldots, a_n$ be a sequence of $n$ values. Each value $a_i$ is independently and randomly chosen from a fixed distribution $\mathcal{D}$ (e.g., uniform distribution from the real interval $[0, 1]$). Let $m_i = \min\{a_1, a_2, \ldots, a_i\}$, i.e., the minimum of the first $i$ values. Let r.v. $Y$ denote the number of times the minimum value is updated, i.e., the number of times $m_i \neq m_{i+1}$. Then $E[Y] = O(\log n)$.

**Solution:** First, without loss of generality we can assume that all the values are distinct, since we are upper bounding $Y$. Indeed, when a value repeats, there won't be any update.

Let the indicator random variable $Y_i$ denote the event that $m_i$ is updated. The value $m_i$ will be updated if $a_{i+1}$ is the minimum value among the first $i + 1$ values. The probability that the above event happens is $\frac{1}{i+1}$. This is because each of the first $i + 1$ values are chosen independently from the same distribution, and assuming that values are distinct, the probability that a particular value, (i.e., $a_{i+1}$) is the minimum is $\frac{1}{i+1}$. Hence $E[Y_i] = Pr(Y_i = 1) = \frac{1}{i+1}$.

We have $Y = \sum_{i=1}^{n-1} Y_i$. Thus, $E[Y] = \sum_{i=1}^{n-1} E[Y_i] = \sum_{i=1}^{n-1} \frac{1}{i+1} \leq H_n = \Theta(\log n)$.

Note that $H_n$ is the harmonic mean, defined as $\sum_{i=1}^{n} 1/i$ and is $\Theta(\log n)$.

### C.7 Useful Types of Random Variables

We define two types of random variables that arise often in probabilistic analysis of algorithms — the binomial and the geometric random variables.
C.7. USEFUL TYPES OF RANDOM VARIABLES

C.7.1 Binomial Random variables

Let $X$ be a 0-1 random variable such that

$$Pr(X = 1) = p, \quad Pr(X = 0) = 1 - p.$$  

Then:

$$E[X] = 1 \cdot p + 0 \cdot (1 - p) = p.$$  

Consider a sequence of $n$ independent Bernoulli trials $X_1, ..., X_n$. Let $X = \sum_{i=1}^{n} X_i$.

Then we say that the random variable $X$ has a Binomial distribution with parameters $n$ and $p$ denoted as $X \sim B(n, p)$. Equivalently, $X$ denotes the number of successes obtained when doing $n$ independent trials where each trial has a probability of success $p$. The probability of getting $k$ successes is given by:

$$Pr(X = k) = \binom{n}{k} p^k (1 - p)^{n-k}.$$  

and the mean is:

$$E[X] = np.$$  

C.7.2 The Geometric Distribution

Assume that an experiment has probability $p$ for success $1 - p$ for failure. How many trials do we need till the first success? The number of trials needed is a random variable and is said to have the geometric distribution. Geometric random variable arises often in probabilistic analysis of algorithms, typically in the following fashion. A randomized step will be shown to succeed with some probability $p$. The quantity that will be interest will be number of such steps needed to get one (or more) success(es).

Thus a random variable $X$ is said to be geometrically distributed if:

$$Pr(X = i) = (1 - p)^{i-1} p.$$  

We say $X$ has a geometric distribution with parameter $p$ $X \sim G(p)$. Clearly, this is a probability distribution, as the probabilities sum to 1:

$$\sum_{i=1}^{\infty} Pr(X = i) = \sum_{i=1}^{\infty} (1 - p)^{i-1} p = \frac{p}{1 - (1 - p)} = 1.$$  

Assume that $X$ get values in $\mathcal{N}$.

$$E[X] = \sum_{i>0} i Pr(X = i) = \sum_{i>1} Pr(X \geq i).$$
Let $X \sim G(p)$. Then, the expectation of $X$ is:

$$E[X] = \sum_{i \geq 1} \text{Prob}(X \geq i) = \sum_{i \geq 1} (1-p)^{i-1} = \frac{1}{1-(1-p)} = \frac{1}{p}.$$ 

The geometric distribution is said to be **memoryless**: For any $k > r$,

$$\Pr(X > k \mid X > r) = \frac{(1-p)^k}{(1-p)^r} = (1-p)^{k-r} = \Pr(X > (k-r)).$$

### C.8 Bounding Deviation of a Random Variable from its Expectation

In many applications, we are interested in bounding the deviation of a random variable from its expectation. This is useful in bounding the probability of success (or failure) of an algorithm. The typical way to do this is to first compute the expectation, and then showing that with good probability the random variable does not deviate “too far” from its expectation, i.e., it is “concentrated” around its mean.

#### C.8.1 Markov Inequality

There are two basic deviation inequalities: Markov and Chebyshev. Markov’s inequality uses only the value of the expectation (also called “first moment”). Hence it can be quite weak.

**Theorem C.2.** [Markov Inequality] For any non-negative random variable and for any $a > 0$

$$\Pr(X \geq a) \leq \frac{E[X]}{a}.$$ 

**Proof.** For any $a > 0$, let $I$ be an indicator r.v. for the event $X \geq a$. Then $I \leq X/a$. Taking expectations on both sides, we get $E[I] = \Pr(X \geq a) \leq E[X]/a$. 

Chebyshev’s inequality gives a stronger bound, which assumes that the variance (or the “second moment”) or **standard deviation** of the random variable is known. These are defined below.
**C.8. BOUNDING DEVIATION OF A RANDOM VARIABLE FROM ITS EXPECTATION**

**Definition C.8.1.** The variance of a random variable $X$ is

$$\text{Var}[X] = E[(X - E[X])^2].$$

The standard deviation of a random variable $X$ is

$$\sigma(X) = \sqrt{\text{Var}[X]}.$$

**Theorem C.3** (Chebyshev’s Inequality). For any random variable

$$\Pr(|X - E[X]| \geq a) \leq \frac{\text{Var}[X]}{a^2}.$$

Proof. \[\Pr(|X - E[X]| \geq a) = \Pr((X - E[X])^2 \geq a^2).\]

By Markov inequality,

$$\Pr((X - E[X])^2 \geq a^2) \leq \frac{E[(X - E[X])^2]}{a^2} = \frac{\text{Var}[X]}{a^2}.$$

\[\square\]

A useful special case of the Markov and Chebyshev’s inequalities is given by the following theorem. This is typically useful in showing whether a non-negative integer-valued random variable takes 0 value or otherwise.

**Theorem C.4** (First and Second Moment Inequalities). Let $X$ be a non-negative integer-valued r.v. Then

$$\Pr(X \geq 1) \leq E(X).$$

$$\Pr(X = 0) \leq \frac{\text{Var}(X)}{(E(X))^2} = \frac{E(X^2)}{(E(X))^2} - 1.$$

**C.8.2 Chernoff Bound for Sum of Indicator R.V’s**

Chernoff bound is a standard tool to show deviation bounds for a random variable that can be expressed as a sum of independent indicator (or Bernoulli) random variables. Such a random variable typically comes in many algorithmic applications. We will state these bounds here and then prove them in Section C.9.1. Typically, in many algorithmic applications, the proofs are not needed to use these bounds.

One way to understand Chernoff bounds is from the point of view of the Binomial random variable $B(n,p)$. As seen from Section C.7.1, the mean of the binomial distribution is $\mu = np$. What is the probability that $B(n,p)$ takes values that are far away from its mean? The probability that $B(n,p)$ takes a value $k > a\mu$, where $a > 1$ is
some parameter is:

\[
\Pr(B(n, p) > \alpha \mu) = \sum_{k=\lceil \alpha \mu \rceil}^{n} \binom{n}{k} p^k (1 - p)^{n-k}.
\]

This probability is called the “upper tail”, i.e., the probability of deviating above the mean. Analogously, one can write an expression for calculating the “lower tail”, i.e., the probability of deviating below the mean. The above formula does not directly give a useful way of quantifying the tail probabilities; as such it is quite a cumbersome sum to manipulate. On the other hand, Chernoff bound approximates the above sum in a way that is very useful to interpret and apply in applications. Moreover, the Binomial tail bounds are a special case of Chernoff, where the success probabilities of all trials are the same (equal to \( p \)); more generally, in Chernoff, each trial can have a different success probability — see Theorem C.5.

**Theorem C.5** (Chernoff bounds). Let \( X_1, X_2, \ldots, X_n \) be independent indicator random variables such that, for \( 1 \leq i \leq n \), \( \Pr[X_i = 1] = p_i \), where \( 0 < p_i < 1 \).

Define \( X = \sum_{i=1}^{n} X_i \) and let \( \mu = E[X] = \sum_{i=1}^{n} p_i \).

1. **Upper Tail Bounds:**
   a. For any \( \delta > 0 \),
   \[
   \Pr(X > (1 + \delta) \mu) < \left( \frac{e^\delta}{(1 + \delta)^{1+\delta}} \right)^\mu
   \]
   b. For \( 0 < \delta < 1 \),
   \[
   \Pr(X > (1 + \delta) \mu) \leq e^{-\mu \delta^2/3}
   \]
   c. For \( R \geq 6 \mu \),
   \[
   \Pr(X \geq R) \leq 2^{-R}
   \]

2. **Lower Tail Bound:**
   For \( 0 < \delta < 1 \),
   \[
   \Pr(X < (1 - \delta) \mu) < e^{-\mu \delta^2/2}
   \]

**C.9 Moment Generating Function**

The moment generating function \( M(t) \) of (discrete) random variable \( X \) is defined for all real values \( t \) by

\[
M(t) = E[e^{tx}] = \sum_x e^{tx} \Pr(X = x).
\]

All moments of \( X \) can be obtained from \( M(t) \) and then evaluating the result at \( t = 0 \), hence the name.

\[
M'(t) = \frac{d}{dt} E[e^{tx}] = E[\frac{d}{dt} (e^{tx})] = E[X^2e^{tx}].
\]
Thus, \( M'(0) = E[X] \) and the \( n \)th derivative of \( M(t) \) is given by:

\[
M^n(t) = E[X^n e^{tX}] \quad n \geq 1
\]

and

\[
M^n(0) = E[X^n] \quad n \geq 1.
\]

**Example: Binomial Distribution** \( B(n, p) \).

We calculate the Moment generating function of the Binomial random variable.

\[
M(t) = E[e^{tX}] = \sum_{k=0}^{n} e^{tk} \binom{n}{k} p^k (1-p)^{n-k} = \sum_{k=0}^{n} \binom{n}{k} (pe^t)^k (1-p)^{n-k} = (pe^t + 1-p)^n.
\]

**Example: Unit Normal Distribution** \( N(0, 1) \). We calculate the Moment generating function of the unit normal distribution.

\[
M(t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{tx} e^{-x^2/2} dx = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-(x-t)^2/2+t^2/2} dx = e^{t^2/2}.
\]

**Important Properties of MGFs.**

1. The moment generating function of the sum of independent random variables equals the product of the individual moment generating functions.
2. MGF uniquely determines the distribution.

**Chernoff Bounds via MGF**

Let \( X \) be a random variable and \( M \) be its MGF. Then, for all \( t > 0 \), via Markov’s inequality, we have:

\[
\Pr(X \geq a) = \Pr(e^{tX} \geq e^{ta}) \leq \frac{E[e^{tX}]}{e^{ta}} = e^{-ta} M(t).
\]

Similarly, for all \( t < 0 \),

\[
\Pr(X \leq a) \leq e^{ta} M(t).
\]

The above bounds are called Chernoff bounds. We obtain the best bound by using the \( t \) that minimizes the right hand side.

**Example.** Let \( X \) be the standard normal r.v. \( P(X \geq a) \leq e^{-ta} e^{t^2/2} \) for all \( t > 0 \). The right hand side is minimized for \( t = a \). Thus, for \( a > 0 \)

\[
P(X \geq a) \leq e^{-a^2/2}.
\]
Similarly, for \( a < 0 \),
\[
\Pr(X \leq a) \leq e^{-a^2/2}.
\]

### C.9.1 Proof of Chernoff Bound for Sum of Indicator R.V's

**Theorem C.6.** Let \( X_1, X_2, \ldots, X_n \) be independent indicator random variables such that, for \( 1 \leq i \leq n \), \( \Pr[X_i = 1] = p_i \), where \( 0 < p_i < 1 \). Then, for \( X = \sum_{i=1}^{n} X_i \), 
\[
\mu = E[X] = \sum_{i=1}^{n} p_i,
\]
and any \( \delta > 0 \),
\[
\Pr(X > (1 + \delta)\mu) < \left( \frac{e^\delta}{1 + \delta} \right)^\mu.
\]

For \( 0 < \delta < 1 \),
\[
\Pr(X > (1 + \delta)\mu) \leq e^{-\mu \delta^2/3}.
\]

For \( R \geq 6\mu \),
\[
\Pr(X \geq R) \leq 2^{-R}.
\]

For \( 0 < \delta < 1 \),
\[
\Pr(X < (1 - \delta)\mu) < e^{-\mu \delta^2/2}.
\]

**Proof.** **Upper tail:** For any positive real \( t \),
\[
\Pr(X > (1 + \delta)\mu) = \Pr(e^{tX} > e^{t(1+\delta)\mu}).
\]

By Markov’s inequality,
\[
\Pr(X > (1 + \delta)\mu) < \frac{E[e^{tX}]}{e^{t(1+\delta)\mu}}
\]
\[
= \frac{E[e^{t \sum_{i=1}^{n} X_i}]}{e^{t(1+\delta)\mu}} = \frac{E[\prod_{i=1}^{n} e^{t X_i}]}{e^{t(1+\delta)\mu}}
\]
\[
= \prod_{i=1}^{n} \left( p_i e^t + 1 - p_i \right) = \prod_{i=1}^{n} (1 + p_i (e^t - 1))
\]
\[
< \prod_{i=1}^{n} e^{p_i (e^t - 1)} = \frac{e^{\sum_{i=1}^{n} p_i (e^t - 1)}}{e^{t(1+\delta)\mu}}
\]
\[
\leq \left( \frac{e^\delta}{1 + \delta} \right)^\mu
\]
for \( t = \ln(1 + \delta) \).

Using \( \delta - (1 + \delta)\ln(1 + \delta) \leq -\delta^2/3 \) for \( 0 < \delta < 1 \) we get
\[
\Pr(X > (1 + \delta)\mu) \leq e^{-\mu \delta^2/3}.
\]
For $R \geq 6\mu, \delta \geq 5$.

\[
Pr(X \geq (1 + \delta)\mu) \leq \left(\frac{e^\delta}{(1 + \delta)^{(1 + \delta)}}\right)^\mu \\
\leq \left(\frac{e}{6}\right)^R \\
\leq 2^{-R}.
\]

**Lower tail:**

\[
Pr(X < (1 - \delta)\mu) = Pr(e^{-tX} > e^{-t(1 - \delta)\mu}).
\]

By Markov's inequality,

\[
Pr(X < (1 - \delta)\mu) < \frac{E[e^{-tX}]}{e^{-t(1 - \delta)\mu}}.
\]

Similar calculations yield

\[
\frac{E[e^{-tX}]}{e^{-t(1 - \delta)\mu}} < \frac{e^{(e^{-t} - 1)\mu}}{e^{-t(1 - \delta)\mu}}.
\]

For $t = \ln(1/(1 - \delta))$, the right hand side of the above is

\[
\leq \left(\frac{e^{-\delta}}{(1 - \delta)^{(1 - \delta)}}\right)^\mu.
\]

Since $(1 - \delta)^{(1 - \delta)} > e^{-\delta + \delta^2/2}$, we have

\[
Pr(X < (1 - \delta)\mu) < e^{-\mu/2}.
\]

\[\square \]

**C.9.2 Example application of Chernoff bound**

**Theorem C.7.** Consider $n$ coin flips, let $X$ be the number of heads,

\[
Pr(|X - \frac{n}{2}| > \frac{1}{2} \sqrt{6n \log n}) \leq \frac{2}{n}
\]

**Proof.**

\[
E[X] = n/2
\]

We need

\[
\frac{n}{2} - \frac{1}{2} \sqrt{6n \log n} \leq X \leq \frac{n}{2} + \frac{1}{2} \sqrt{6n \log n}
\]

or

\[
X = \frac{n}{2} (1 \pm \sqrt{\frac{6 \log n}{n}})
\]
Fixing $\delta = \sqrt{\frac{\log n}{n}}$

\[
\Pr(X < (1 - \delta)n/2) \leq e^{-\frac{\delta^2}{2}} \leq 1/n
\]

\[
\Pr(X > (1 + \delta)n/2) \leq e^{-\frac{\delta^2}{2}} \leq 1/n
\]

\[\Box\]

**Comparing Markov, Chebyshev, and Chernoff.**

What is the probability of more than $\frac{3N}{4}$ heads in $N$ coin flips?

1. Using Markov Inequality:

\[
Pr(X \geq 3N/4) \leq 2/3.
\]

2. Using Chebyshev’s Inequality:

\[
Pr(X \geq 3N/4) \leq \frac{4}{N}.
\]

3. Using the Chernoff bound:

\[
Pr(X \geq 3N/4) \leq e^{-\frac{N}{2} + 1}.
\]

We see that Chernoff gives a stronger bound compared to Chebyshev which in turn is stronger compared to Markov. The reason is Markov’s inequality uses only the first moment (i.e., the expectation) of the random variable in its bound, whereas, Chebyshev uses the first and second moment (equivalently, the variance) into account, while Chernoff uses all the moments (via the MGF).

**C.10 Conditional Expectation**

The conditional expectation of a random variable is defined as

\[
E[Y|Z = z] = \sum_y \Pr(Y = y | Z = z)
\]

where the summation is over all $y$ in the range of $Y$.

A useful identity is:

\[
E[X] = \sum_y \Pr(Y = y)E[X|Y = y]
\]

The expression $E[Y|Z]$ is a random variable $f(Z)$ that takes the value $E[Y|Z = z]$ when $Z = z$. 
Theorem C.8.

\[ E[X] = E[E[X | Y]]. \]
Appendix D

Graph-theoretic Concepts

This appendix reviews basic concepts in graph theory. For more details on these concepts see [3].

D.1 Definition

We formally define a graph as follows.

**Definition D.1.1.** An undirected graph $G$ is a pair $(V, E)$ where

- $V$ is the set of vertices.
- $E \subseteq V^2$ is the set of edges (unordered pairs) $E = \{(u, v) \mid u, v \in V\}$.

In a directed graph the edges have directions, i.e., ordered pairs — $(u, v)$ means that $u$ points to $v$ and $(v, u)$ means that $v$ points to $u$. A weighted graph includes a weight function $w : E \rightarrow Q$ attaching a (rational) number (weight) to each edge.

Running time of a graph algorithm on a graph $G = (V, E)$ is usually in terms of $|V| = n$ and $|E| = m$. (We will use this notation throughout.) Sometimes, the running time can be a function of weights on the edges.

Figure D.1 shows an example of an undirected graph and an undirected weighted graph. Note that there can be at most $\binom{n}{2}$ edges in an undirected graph with $n$ nodes.

D.2 Path, Cycle, and Tree

These are fundamental structures in graphs and are defined as follows.
APPENDIX D. GRAPH-THEORETIC CONCEPTS

**Figure D.1**: A connected undirected graph (left) and the same graph with weights (right).

**Definition D.2.1.** A *path* in a graph $G = (V, E)$ is a sequence of vertices $v_1, v_2, ..., v_k$ such that for $1 \leq i \leq k-1$, $(v_i, v_{i+1}) \in E$.

A *cycle* in a graph $G = (V, E)$ is a path $v_1, v_2, ..., v_k$ such that $(v_k, v_1) \in E$.

A *tree* is a graph with no cycles. A disjoint (i.e., the vertices are disjoint) collection of trees is a *forest*.

**D.3 Distance**

Given a graph $G = (V, E)$, and two vertices $u$ and $v$ belonging to $V$, and a path $P$ between $u$ and $v$, the *length* of the path is the number of edges between $u$ and $v$ in $P$. The *distance* between $u$ and $v$ is the length of the *shortest path* between $u$ and $v$. The shortest path between $u$ and $v$ will have *minimum length* among all other paths between the two vertices.

**D.4 Subgraph and Induced Subgraph**

**Definition D.4.1.** A graph $H = (V', E')$ is a *subgraph* of $G = (V, E)$ iff $V' \subseteq V$ and $E' \subseteq E$.

If $V' = V$, then $H$ is a *spanning* subgraph.

Given a subset $U \subseteq V$ of vertices, the subgraph *induced* by $U$ in $G$ is the subgraph consisting of all vertices in $U$ and all the edges between them in $G$.

Figure D.4 shows an example of a graph and an induced subgraph.
D.5 Degree of a vertex

Given an undirected graph \( G(V, E) \), the degree of a vertex is the number of its neighbors, i.e., the number of vertices that it shares an edge with. The degree of \( v \) is denoted by \( \text{deg}(v) \).

**Lemma D.1.** In an undirected graph \( G = (V, E) \): \( \sum_{v \in V} \text{deg}(v) = 2|E| = 2m. \)

**Proof.** Consider an edge \((u, v) \in E\). This contributes one to \( \text{deg}(u) \) and one to \( \text{deg}(v) \). \( \square \)

D.6 Connectivity

Connectivity is a basic concept in graph theory.

**Definition D.6.1.** An undirected graph is **connected** if there is a path between every pair of vertices. A directed graph is **connected** if there is a directed path from \( u \) to \( v \) or a directed path from \( v \) to \( u \), for every pair of vertices \( u, v \).

A directed graph is **strongly connected** if there is a directed path from \( u \) to \( v \) and a directed path from \( v \) to \( u \), for every pair of vertices \( u, v \).

A directed graph is **weakly connected** if the undirected graph obtained by ignoring the directions of the edges is connected.

A **connected component** of an undirected graph \( G \) is a maximal induced subgraph of \( G \) that is connected. It is maximal in the sense that no more vertices can be added to the subgraph, while still keeping the subgraph connected. Figure D.6 shows the example of a disconnected graph.
Figure D.3: A disconnected graph.

D.7 Spanning Tree (ST)

Definition D.7.1. Given a connected graph \( G = (V, E) \) a spanning tree \( T \) in \( G \) is a spanning subgraph of \( G \) which is a tree.

If the graph is not connected, then we can define a spanning tree on each connected component. A (disjoint) collection of spanning trees is called a spanning forest.

Maximality of a Spanning Tree. Note that a spanning tree \( T \) is “maximal" in the sense that you cannot add additional edges — any edge added to \( T \) will close a cycle.

The reason of the above is as follows. Assume that \( T \) is a spanning tree in \( G \), and we add an edge \((v, u)\) to \( T \). There is a path \( p \) from \( u \) to \( v \) using only edges of \( T \) (since it’s a spanning tree). Thus, the path \( p \) and the edge \((v, u)\) must include a cycle.

D.7.1 The Size of a Spanning Tree

Theorem D.1. A spanning tree of a connected graph \( G = (V, E) \) has \(|V| - 1 = n - 1\) edges.

Proof. The proof is by induction on \(|V|\). Base case \(|V| = 1\) is trivial. Assume that the lemma is true for all graphs of size \(< |V|\). Consider a connected graph \( G \) of size \(|V|\) and a spanning tree \( T \) of \( G \). Remove an edge \((u, v)\) of \( T \). This breaks \( T \) into two components — \( T_1 \) and \( T_2 \). \( T_1 \) (\( T_2 \)) is a spanning tree of the subgraph induced by the vertices of \( T_1 \) (\( T_2 \)). The number of edges in \( T \) is \(|T_1| - 1 + |T_2| - 1 + 1 = |T| - 1 = |V| - 1\).

D.8 Graph Search Algorithms

There are two fundamental algorithms to search (explore) a graph: Depth-first search (DFS) and Breadth-first search (BFS). While DFS explores graph by always exploring
the most recently visited vertex, BFS explores nodes in increasing distances from a given source node.

**D.8.1 Breadth-first search (BFS)**

In BFS, the search starts from a vertex $s$, and visits vertices in increasing order of distance from $s$. BFS yields a spanning tree called breadth-first spanning (BFS) tree in an undirected connected graph. An important application of breadth-first search is finding the (shortest path) distance (as well as shortest path) from $s$ to all other vertices. Recall that the distance between two vertices is the number of edges in a shortest path between them.

Figure D.4 shows an undirected graph and its BFS tree.

**D.8.2 Depth-first search (DFS)**

In DFS, the search starts from a vertex $s$, and edges are explored out of the most recently visited vertex. DFS yields a spanning tree called depth-first spanning (DFS) tree. Figure D.5 shows a graph and its DFS tree.
Figure D.5: An undirected graph and its DFS tree rooted at vertex a where the DFS is started.
Appendix E

Commonly Used Math Formulas in Algorithm Analysis

We collect commonly used math formulas in algorithm analysis. For a elaborate list see the theoretical computer cheat sheet by Steve Seiden (https://www.tug.org/texshowcase/cheat.pdf)

E.1 Geometric series

Summing a geometric series occurs very often in algorithmic analysis. The formula for the sum of a geometric series:

\[\sum_{i=0}^{k-1} x^i = 1 + x + \cdots + x^{k-1} = \frac{x^k - 1}{x - 1}.\]

where, \(x\) is called the (common) ratio.

Thus:
\[\sum_{k=0}^{n} x^k = \Theta(x^n) \text{ if } x > 1 \text{ (increasing series).}\]
\[\sum_{k=0}^{n} x^k = \Theta(1) \text{ if } x < 1 \text{ (decreasing series).}\]
If the series is infinite and \(|x| < 1\), then

\[\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}.\]

E.2 Combinatorial Inequalities

\[\binom{n}{k}^k \leq \left(\frac{n}{k}\right)^k \leq \left(\frac{ne}{k}\right)^k.\]
APPENDIX E. COMMONLY USED MATH FORMULAS IN ALGORITHM ANALYSIS
Bibliography


