RESILIENT NETWORKS

Mathematical Framework

Pushkin Kachroo
Shankar Sastry

University of California Berkeley

© Pushkin Kachroo
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>List of Figures</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>List of Tables</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>Introduction to Resilient Networks</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>Algorithms and Complexity Analysis</td>
<td>13</td>
</tr>
<tr>
<td>2.1</td>
<td>Complexity Notation</td>
<td>13</td>
</tr>
<tr>
<td>2.1.1</td>
<td>Big-( O ) Algebra</td>
<td>15</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Comparing Orders</td>
<td>17</td>
</tr>
<tr>
<td>2.2</td>
<td>Complexity Examples</td>
<td>17</td>
</tr>
<tr>
<td>2.2.1</td>
<td>( O(1) ) Example</td>
<td>17</td>
</tr>
<tr>
<td>2.2.2</td>
<td>( O(\log(n)) ) Example: Binary Search</td>
<td>18</td>
</tr>
<tr>
<td>2.2.3</td>
<td>( O(n) ) Example: Linear Search</td>
<td>18</td>
</tr>
<tr>
<td>2.2.4</td>
<td>( O(n^2) ) Example: Bubble Sort</td>
<td>20</td>
</tr>
<tr>
<td>2.2.5</td>
<td>( O(n^p) ) Polynomial Complexity</td>
<td>21</td>
</tr>
<tr>
<td>2.2.6</td>
<td>( O(2^n) ) Exponential Complexity</td>
<td>22</td>
</tr>
<tr>
<td>2.2.7</td>
<td>( O(n!) ) Factorial Complexity</td>
<td>22</td>
</tr>
<tr>
<td>2.2.8</td>
<td>( O(n^n) ) Complexity</td>
<td>22</td>
</tr>
<tr>
<td>2.3</td>
<td>Sorting Algorithms</td>
<td>23</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Bubble Sort</td>
<td>23</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Selection Sort</td>
<td>24</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Insertion Sort</td>
<td>25</td>
</tr>
<tr>
<td>2.3.4</td>
<td>Merge Sort</td>
<td>26</td>
</tr>
<tr>
<td>2.3.5</td>
<td>Quick Sort</td>
<td>26</td>
</tr>
<tr>
<td>2.4</td>
<td>Complexity Analysis Example</td>
<td>28</td>
</tr>
<tr>
<td>2.4.1</td>
<td>Median Computation Algorithm Analysis</td>
<td>30</td>
</tr>
<tr>
<td>2.4.2</td>
<td>Divide and Conquer Algorithm Analysis</td>
<td>31</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>2.4.3</td>
<td>Complexity Analysis using Recursive Computation Equation</td>
<td>32</td>
</tr>
<tr>
<td>2.4.4</td>
<td>Average Case Analysis: Randomizing Algorithms</td>
<td>34</td>
</tr>
<tr>
<td>2.4.5</td>
<td>Example of an Average Case Analysis</td>
<td>38</td>
</tr>
<tr>
<td>2.5</td>
<td>Generating Functions Method to Solve Recursive Equations</td>
<td>42</td>
</tr>
<tr>
<td>2.6</td>
<td>Average Case Analysis of Algorithms</td>
<td>42</td>
</tr>
<tr>
<td>2.7</td>
<td>Bibliographical Notes</td>
<td>42</td>
</tr>
<tr>
<td>2.8</td>
<td>Exercises</td>
<td>42</td>
</tr>
<tr>
<td>3</td>
<td>Formal Languages, Grammar, and Machines</td>
<td>43</td>
</tr>
<tr>
<td>3.1</td>
<td>Languages, Grammars, and Machines</td>
<td>43</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Language</td>
<td>43</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Grammar</td>
<td>45</td>
</tr>
<tr>
<td>3.1.3</td>
<td>Machine</td>
<td>47</td>
</tr>
<tr>
<td>3.2</td>
<td>Finite State Automata, Regular Languages, and Grammars</td>
<td>48</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Finite State Automata</td>
<td>48</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Finite State Automata as Transducers</td>
<td>51</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Regular Languages</td>
<td>55</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Regular Grammars</td>
<td>55</td>
</tr>
<tr>
<td>3.2.5</td>
<td>Closure Properties of Regular Languages</td>
<td>56</td>
</tr>
<tr>
<td>3.2.6</td>
<td>Decidable Properties of Regular Languages</td>
<td>57</td>
</tr>
<tr>
<td>3.2.7</td>
<td>A Nonregular Language</td>
<td>57</td>
</tr>
<tr>
<td>3.3</td>
<td>Pushdown Automata, Context-Free Languages, and Grammars</td>
<td>59</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Pushdown Automata</td>
<td>60</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Context-Free Grammars and Languages</td>
<td>63</td>
</tr>
<tr>
<td>3.3.3</td>
<td>A Non Context Free Language</td>
<td>68</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Closure Properties of Context Free Languages</td>
<td>70</td>
</tr>
<tr>
<td>3.3.5</td>
<td>Decidable Properties of Context Free Languages</td>
<td>71</td>
</tr>
<tr>
<td>3.4</td>
<td>Turing Machines and Languages</td>
<td>72</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Turing Machines</td>
<td>72</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Turing Decidable and Semi-decidable Languages</td>
<td>82</td>
</tr>
<tr>
<td>3.5</td>
<td>Chomsky Hierarchy and Language Containments</td>
<td>86</td>
</tr>
<tr>
<td>3.6</td>
<td>Bibliographical Notes</td>
<td>87</td>
</tr>
<tr>
<td>3.7</td>
<td>Exercises</td>
<td>87</td>
</tr>
<tr>
<td>4</td>
<td>Computation Theory</td>
<td>89</td>
</tr>
<tr>
<td>4.1</td>
<td>Equivalent Turing Machines</td>
<td>89</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Turing Machine with Stay</td>
<td>89</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Multitrack Turing Machine</td>
<td>90</td>
</tr>
<tr>
<td>4.1.3</td>
<td>One-way Tape Turing Machine</td>
<td>90</td>
</tr>
<tr>
<td>4.1.4</td>
<td>Multi-Tape Turing Machine</td>
<td>90</td>
</tr>
</tbody>
</table>
CONTENTS

4.1.5 Read-only Tape (Off-line) Turing Machine .......... 91
4.1.6 Multidimensional Turing Machine .................. 91
4.1.7 Nondeterministic Turing Machine ................... 91
4.1.8 Universal Turing Machine ............................ 93

4.2 Decidability and Computability .......................... 93
4.2.1 Computability .................................... 94
4.2.2 Decidable Problems ................................. 98
4.2.3 Halting Problem .................................. 98
4.2.4 Undecidable Problems via Reducibility ............... 101
4.2.5 Many-one Reduction ............................... 105

4.3 Complexity ........................................ 106
4.3.1 Kolmogorov Complexity ............................. 106
4.3.2 Compressibility of Strings .......................... 107

4.4 Time Complexity .................................... 107
4.4.1 Time Complexity of Different Types of Turing Machines 109
4.4.2 P and NP Complexity ............................... 111
4.4.3 Space Complexity ................................ 119

4.5 Bibliographical Notes .................................. 126
4.6 Exercises ........................................... 126

5 Graphs and Trees ...................................... 127
5.1 Graphs ............................................. 127
5.1.1 Types of Graphs ................................ 128
5.1.2 Some Graph Properties ............................ 130
5.1.3 Set Operations on Graphs ........................... 134
5.1.4 Ambulation on Graphs ............................. 134

5.2 Bibliographical Notes ................................ 134
5.3 Exercises ........................................... 134

6 Mathematical Programming and Optimization ............. 135
6.1 Introduction ....................................... 135
6.2 Bibliographical Notes ................................ 135
6.3 Exercises ........................................... 135

7 Queueing Networks .................................... 137
7.1 Introduction ....................................... 137
7.2 Bibliographical Notes ................................ 137
7.3 Exercises ........................................... 137

8 Network Calculus .................................... 139
8.1 Introduction ....................................... 139
<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.2 Bibliographical Notes</td>
<td>139</td>
</tr>
<tr>
<td>8.3 Exercises</td>
<td>139</td>
</tr>
<tr>
<td>9 Game Theory on Networks</td>
<td>141</td>
</tr>
<tr>
<td>9.1 Introduction</td>
<td>141</td>
</tr>
<tr>
<td>9.2 Bibliographical Notes</td>
<td>141</td>
</tr>
<tr>
<td>9.3 Exercises</td>
<td>141</td>
</tr>
<tr>
<td>10 Network Dynamics and Control</td>
<td>143</td>
</tr>
<tr>
<td>10.1 Introduction</td>
<td>143</td>
</tr>
<tr>
<td>10.2 Bibliographical Notes</td>
<td>143</td>
</tr>
<tr>
<td>10.3 Exercises</td>
<td>143</td>
</tr>
<tr>
<td>11 Complex Networks</td>
<td>145</td>
</tr>
<tr>
<td>11.1 Introduction</td>
<td>145</td>
</tr>
<tr>
<td>11.2 Bibliographical Notes</td>
<td>145</td>
</tr>
<tr>
<td>11.3 Exercises</td>
<td>145</td>
</tr>
<tr>
<td>12 Dynamic Traffic Assignment: A Survey of Mathematical Models and</td>
<td>147</td>
</tr>
<tr>
<td>Techniques</td>
<td></td>
</tr>
<tr>
<td>12.1 Introduction</td>
<td>147</td>
</tr>
<tr>
<td>12.2 Mathematical Programming for Static Traffic Assignment</td>
<td>149</td>
</tr>
<tr>
<td>12.2.1 User-equilibrium</td>
<td>150</td>
</tr>
<tr>
<td>12.2.2 System Optimal Solution</td>
<td>152</td>
</tr>
<tr>
<td>12.2.3 Numerical Schemes</td>
<td>153</td>
</tr>
<tr>
<td>12.3 Variational Inequality based Static Traffic Assignment Model</td>
<td>154</td>
</tr>
<tr>
<td>12.4 Projected Dynamical Systems: Dynamic Variational Equation Model</td>
<td>158</td>
</tr>
<tr>
<td>12.4.1 Dynamic Route Choice</td>
<td>159</td>
</tr>
<tr>
<td>12.5 Dynamic Traffic Assignment</td>
<td>160</td>
</tr>
<tr>
<td>12.5.1 Dynamic Traffic Assignment: Discrete Time</td>
<td>160</td>
</tr>
<tr>
<td>12.5.2 Dynamic Traffic Assignment: Continuous Time</td>
<td>161</td>
</tr>
<tr>
<td>12.6 Travel Time and FIFO Issue</td>
<td>162</td>
</tr>
<tr>
<td>12.7 Macroscopic Model for DTA</td>
<td>163</td>
</tr>
<tr>
<td>12.7.1 Greenshield's Model</td>
<td>163</td>
</tr>
<tr>
<td>12.7.2 Generalized/Weak Solution for the LWR Model</td>
<td>164</td>
</tr>
<tr>
<td>12.7.3 Scalar Initial-Boundary Problem</td>
<td>166</td>
</tr>
<tr>
<td>12.7.4 Macroscopic (PDE) Traffic Network</td>
<td>167</td>
</tr>
<tr>
<td>12.7.5 Travel Time Dynamics</td>
<td>169</td>
</tr>
<tr>
<td>12.8 Simulation based DTA</td>
<td>170</td>
</tr>
<tr>
<td>12.8.1 Iterations for User Equilibrium</td>
<td>170</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

2.1 O(1) Example: Add First Two Items .................................. 18
2.2 Binary Search ............................................................. 19
2.3 Linear Search ............................................................. 21
2.4 Bubble Sort ................................................................. 23
2.5 Selection Sort ............................................................... 24
2.6 Insertion Sort ............................................................... 25
2.7 Merge Sort ................................................................. 27
2.8 Quick Sort ................................................................. 28
2.9 Analysis Tree for Merge Sort ............................................ 32
2.10 Turing Machine Input and Output ...................................... 39
2.11 Turing Machine Transitions ............................................ 39
2.12 Binary Addition ........................................................... 39

3.1 Finite Automata Example ................................................. 47
3.2 DFA: (ab)^n Acceptor ...................................................... 49
3.3 Automaton with Input Tape .............................................. 50
3.4 Nondeterministic Finite Automata Example .......................... 51
3.5 Mealy Machine ............................................................. 52
3.6 Moore Machine ............................................................ 53
3.7 Pumping Lemma for Regular Languages .............................. 58
3.8 PDA: a^n b^n Acceptor .................................................... 60
3.9 Pushdown Automaton with Input Tape and Stack .................. 61
3.10 Two Derivation Trees for x + y * z .................................. 64
3.11 The Only Derivation Tree for x + y * z .............................. 66
3.12 Pumping Lemma for Context Free Languages ..................... 69
3.13 Comparing DFA, NPDA, and TM ..................................... 73
3.14 Comparing Transition Graphs for DFA, NPDA, and TM ......... 73
3.15 Transition by a Turing Machine ....................................... 75
3.16 Turing Machine Transition ............................................. 75
3.17 Turing Machine Not Halting .......................................... 76
3.18 Turing Machine Not Halting Transition Graph ..................... 76
<table>
<thead>
<tr>
<th>Figure Number</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.19</td>
<td>Set of Natural Numbers</td>
<td>79</td>
</tr>
<tr>
<td>3.20</td>
<td>Set of Even Natural Numbers</td>
<td>80</td>
</tr>
<tr>
<td>3.21</td>
<td>Set of Integers going to Infinity on both Ends</td>
<td>80</td>
</tr>
<tr>
<td>3.22</td>
<td>Set of Integers showing Bijection</td>
<td>80</td>
</tr>
<tr>
<td>3.23</td>
<td>Bijection for the Set of Positive Rationals</td>
<td>81</td>
</tr>
<tr>
<td>3.24</td>
<td>Cantor's Diagonalization</td>
<td>81</td>
</tr>
<tr>
<td>3.25</td>
<td>Turing Semidecidable Language Enumeration</td>
<td>83</td>
</tr>
<tr>
<td>4.1</td>
<td>Multitrack Turing Machine</td>
<td>90</td>
</tr>
<tr>
<td>4.2</td>
<td>Twotrack One-way Turing Machine and Single track Two-way</td>
<td>91</td>
</tr>
<tr>
<td>4.3</td>
<td>Multitape Turing Machine</td>
<td>91</td>
</tr>
<tr>
<td>4.4</td>
<td>Turing Machine with Read-only Tape</td>
<td>92</td>
</tr>
<tr>
<td>4.5</td>
<td>Multidimensional Turing Machine</td>
<td>92</td>
</tr>
<tr>
<td>4.6</td>
<td>Nondeterministic Turing Machine</td>
<td>93</td>
</tr>
<tr>
<td>4.7</td>
<td>Reduction to Blank Tape Halting Problem</td>
<td>102</td>
</tr>
<tr>
<td>4.8</td>
<td>Reduction to State Entry Problem</td>
<td>102</td>
</tr>
<tr>
<td>4.9</td>
<td>Reduction to Membership Problem</td>
<td>103</td>
</tr>
<tr>
<td>4.10</td>
<td>Reduction to Empty Language Problem</td>
<td>104</td>
</tr>
<tr>
<td>4.11</td>
<td>Reduction to Nontrivial Property</td>
<td>105</td>
</tr>
<tr>
<td>4.12</td>
<td>Multitape to Single Tape</td>
<td>109</td>
</tr>
<tr>
<td>4.13</td>
<td>Nondeterministic to Deterministic Turing Machine</td>
<td>110</td>
</tr>
<tr>
<td>4.14</td>
<td>Read-only,Write-only,Read-write Tape Turing Machine</td>
<td>123</td>
</tr>
<tr>
<td>4.15</td>
<td>Sublinear Simulation</td>
<td>124</td>
</tr>
<tr>
<td>4.16</td>
<td>Directed Graph Path problem</td>
<td>125</td>
</tr>
<tr>
<td>4.17</td>
<td>Breadth First Algorithm for PATH</td>
<td>126</td>
</tr>
<tr>
<td>5.1</td>
<td>Some Example Graphs</td>
<td>128</td>
</tr>
<tr>
<td>5.2</td>
<td>Some Example Digraphs</td>
<td>128</td>
</tr>
<tr>
<td>5.3</td>
<td>Simple and Non-Simple Graph</td>
<td>129</td>
</tr>
<tr>
<td>5.4</td>
<td>Simple and Non-Simple Graph</td>
<td>130</td>
</tr>
<tr>
<td>5.5</td>
<td>Hypergraph</td>
<td>130</td>
</tr>
<tr>
<td>5.6</td>
<td>Labelled and a Unlabelled Graph</td>
<td>131</td>
</tr>
<tr>
<td>5.7</td>
<td>Open and Closed Neighborhoods of a Vertex</td>
<td>131</td>
</tr>
<tr>
<td>5.8</td>
<td>Open and Closed Neighborhoods of a Vertex Set</td>
<td>132</td>
</tr>
<tr>
<td>5.9</td>
<td>Boundary of a Vertex Set</td>
<td>132</td>
</tr>
<tr>
<td>5.10</td>
<td>Digraphs and Oriented Graphs of Order 3 and Size 3</td>
<td>133</td>
</tr>
<tr>
<td>5.11</td>
<td>Graph Degrees</td>
<td>133</td>
</tr>
<tr>
<td>12.1</td>
<td>Sample Network</td>
<td>150</td>
</tr>
<tr>
<td>12.2</td>
<td>BPR Link Performance Function</td>
<td>151</td>
</tr>
<tr>
<td>12.3</td>
<td>Variational Inequality</td>
<td>155</td>
</tr>
</tbody>
</table>
List of Figures

12.4 Minimizer in the Interior ................................................. 156
12.5 Minimizer on the Boundary .............................................. 156
12.6 Violation of Kuhn-Tucker Condition .................................. 158
12.7 The vector field .............................................................. 159
12.8 FIFO Violation ................................................................. 163
12.9 Fundamental Diagram using Greenshield Model ...................... 164
12.10 Traffic Characteristics ..................................................... 165
12.11 Boundary Data ............................................................... 167
12.12 Traffic Node with Incoming and Outgoing Links .................... 167
12.13 Travel Time on a Link ....................................................... 169
# List of Tables

<table>
<thead>
<tr>
<th>Section</th>
<th>Table Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Grammar for Example 3.4</td>
<td>45</td>
</tr>
<tr>
<td>3.2</td>
<td>Right Linear Grammar</td>
<td>56</td>
</tr>
<tr>
<td>3.3</td>
<td>Left Linear Grammar</td>
<td>56</td>
</tr>
<tr>
<td>3.4</td>
<td>NPDA Accepting $a^n b^n$</td>
<td>62</td>
</tr>
<tr>
<td>3.5</td>
<td>Context-free Grammar</td>
<td>63</td>
</tr>
<tr>
<td>3.6</td>
<td>Grammar for $a^n b^n$</td>
<td>63</td>
</tr>
<tr>
<td>3.7</td>
<td>Grammar for Arithmetic Expressions</td>
<td>65</td>
</tr>
<tr>
<td>3.8</td>
<td>Non-ambiguous Grammar for Arithmetic Expressions</td>
<td>66</td>
</tr>
<tr>
<td>3.9</td>
<td>Unrestricted Grammar</td>
<td>84</td>
</tr>
<tr>
<td>3.10</td>
<td>Context Sensitive Grammar</td>
<td>85</td>
</tr>
<tr>
<td>3.11</td>
<td>Grammar for Example 3.4</td>
<td>87</td>
</tr>
<tr>
<td>4.1</td>
<td>Boolean $or$</td>
<td>112</td>
</tr>
<tr>
<td>4.2</td>
<td>Boolean $and$</td>
<td>112</td>
</tr>
<tr>
<td>4.3</td>
<td>Boolean $not$</td>
<td>113</td>
</tr>
<tr>
<td>4.4</td>
<td>$(A + B) \cdot C$</td>
<td>113</td>
</tr>
<tr>
<td>4.5</td>
<td>Verification of $(A + B) \cdot C$</td>
<td>116</td>
</tr>
<tr>
<td>4.6</td>
<td>Variables for Representation</td>
<td>118</td>
</tr>
<tr>
<td>4.7</td>
<td>Clauses Collection</td>
<td>118</td>
</tr>
<tr>
<td>5.1</td>
<td>Graph Degrees</td>
<td>133</td>
</tr>
<tr>
<td>12.1</td>
<td>Network Notation</td>
<td>149</td>
</tr>
</tbody>
</table>
In this chapter ···
ALGORITHMS AND COMPLEXITY ANALYSIS

This chapter reviews the fundamentals of algorithmic complexity which will be later used to study the various algorithms for networks. The chapter presents the notation for expressing complexity and then presents the theory of languages and machines so that complexity can be stated in terms of Turing machines.

2.1 Complexity Notation

Complexity of an algorithm is analyzed in terms of time complexity and space complexity. Time complexity is obtained in terms of number of computations required, and space in terms of how much memory is required. We are interested not in the exact computation cost, but only the asymptotic order of complexity. It will be assessed as the size of the problem goes to infinity and we want to find out the dominating term of the complexity.

The big-O notation provides an asymptotic upper bound to a function, the big-Ω provides an asymptotic lower bound, and the Θ notation provides an asymptotic tight bound.

Definition 2.1. Big-O Notation: The function $f : \mathbb{N} \rightarrow \mathbb{R}$ has order at most $g$, i.e. $f(n) \in O(g(n))$ if $\exists M > 0$ such that $\exists N > 0, N \in \mathbb{N}$ so that $\forall n > N, |f(n)| \leq M|g(n)|$. □
**Definition 2.2. Big-Ω Notation:** The function \( f : \mathbb{N} \to \mathbb{R} \) has order at least \( g \), i.e. \( f(n) \in \Omega(g(n)) \) if \( \exists M > 0 \) such that \( \forall N > 0, N \in \mathbb{N} \) so that \( \forall n > N, |f(n)| \geq M|g(n)| \). □

**Definition 2.3. Θ Notation:** The function \( f : \mathbb{N} \to \mathbb{R} \) has order the same as \( g \), i.e. \( f(n) \in \Theta(g(n)) \) if \( \exists M_1 > 0, M_2 > 0 \) such that \( \exists N > 0, N \in \mathbb{N} \) so that \( \forall n > N, M_1|g(n)| \leq |f(n)| \leq M_2|g(n)| \). □

**Definition 2.4. Little-o Notation:** The function \( f : \mathbb{N} \to \mathbb{R} \) has order smaller than \( g \), i.e. \( f(n) \in o(g(n)) \) if \( \forall \epsilon > 0, \exists N > 0, N \in \mathbb{N} \) such that \( \forall n > N, |f(n)| \leq \epsilon |g(n)| \). □

**Definition 2.5. Little-ω Notation:** The function \( f : \mathbb{N} \to \mathbb{R} \) has order bigger than \( g \), i.e. \( f(n) \in \omega(g(n)) \) if \( \forall \epsilon > 0, \exists N > 0, N \in \mathbb{N} \) such that \( \forall n > N, |f(n)| \geq \epsilon |g(n)| \). □

The definition for big-\( O \) implies that asymptotically the ratio of the two functions is finite. More precisely,

\[
f(n) \in O(g(n)) \rightarrow \limsup_{n \to \infty} \left| \frac{g(n)}{f(n)} \right| < \infty \quad (2.1)
\]

The definition for little-\( o \) implies that asymptotically the ratio of the two functions is zero. More precisely, if \( \exists N > 0 \), such that \( \forall n > Ng(n) \neq 0 \), then

\[
f(n) \in o(g(n)) \Rightarrow \lim_{n \to \infty} \left| \frac{g(n)}{f(n)} \right| = 0 \quad (2.2)
\]

The relationship between the big-\( O \) and the little-\( o \) is similar to the relationship between the big-\( Ω \) and the little-\( ω \) notation.

The definition for little-\( ω \) implies that asymptotically the ratio of the two functions goes to infinity. More precisely, if \( \exists N > 0 \), such that \( \forall n > N g(n) \neq 0 \), then

\[
f(n) \in \omega(g(n)) \Rightarrow \lim_{n \to \infty} \left| \frac{g(n)}{f(n)} \right| = \infty \quad (2.3)
\]

The technique to manipulate the complexity orders is to only consider the highest order terms in the polynomials, and also to ignore the coefficients. For instance we can show from the definitions that the following are true.

**Example 2.6.**

\[
\begin{align*}
O(n^3 + 3n^2 - n + 10) &= O(n^3) \\
O(5n^3) &= O(n^3) \\
O(n \log n + 5n) &= O(n \log n)
\end{align*}
\]
2.1. COMPLEXITY NOTATION

Let us look at some details of this analysis. We will consider the function \( f(n) = 5n^3 + 3n^2 - n + 10 \). In order to compare its order with that of \( g(n) = n^3 \), we take the ratio and the limit and obtain

\[
\lim_{n \to \infty} \frac{5n^3 + 3n^2 - n + 10}{n^3} = 5 < \infty
\]  

(2.4)

Now if we take function \( h(n) = n^4 \), we obtain for the order of \( f(n) \)

\[
\lim_{n \to \infty} \frac{5n^3 + 3n^2 - n + 10}{n^4} = 0 < \infty
\]  

(2.5)

This shows that \( f(n) \in O(g(n)), f(n) \in O(h(n)) \), and we also have \( f(n) \in o(h(n)) \), but \( f(n) \notin o(g(n)) \). In fact what we see is that \( \forall f, g, f(n) \in o(g(n)) \Rightarrow f(n) \in O(g(n)) \), but not the other way around necessarily because for the Little-o, the constant must be zero for the function also to belong to Big-O.

**Example 2.7.** We are given the following functions.

\[
\begin{align*}
  f_1(n) &= 5n^3 + 3n^2 - n + 10 \\
  f_2(n) &= 5n^4 + n \\
  f_3(n) &= 10n^3 + 5n^2
\end{align*}
\]

For these functions we have

\[
\begin{align*}
  f_1(n) &\in o(f_2(n)) \\
  f_1(n) &\in O(f_2(n)) \\
  f_1(n) &\in O(f_3(n)) \\
  f_2(n) &\in \Omega(f_1(n)) \\
  f_1(n) &\in \Theta(f_3(n))
\end{align*}
\]

2.1.1 Big-O Algebra

The Big-O algebra behaves different than the algebra of real numbers as is clear in the following examples.
Example 2.8. We are given the following functions.

\[
\begin{align*}
    f_1(n) &= 2n^3 - n + 1 \\
    f_2(n) &= 5n^2 + 5
\end{align*}
\]

We have for multiplication

\[
O(f_1(n)f_2(n)) = O((2n^3 - n + 1)(5n^2 + 5)) = O(10n^5 + \cdots) = O(n^5)
\]

For addition, we have

\[
O(f_1(n) + f_2(n)) = O((2n^3 - n + 1) + (5n^2 + 5)) = O(2n^3 + \cdots) = O(n^3) = O(f_1(n))
\]

For scaling, we have

\[
O(5 \cdot f_1(n)) = O(5 \cdot (2n^3 - n + 1)) = O(10n^3 + \cdots) = O(n^3) = O(f_1(n))
\]

The Big-O algebra has the following properties.

**Multiplication:**

\[
\begin{align*}
\text{Action} & \quad f_1(n) \in O(g_1(n)), \ f_2(n) \in O(g_2(n)) \Rightarrow f_1f_2(n) \in O(g_1g_2) \\
\text{Absorption} & \quad f \cdot O(g(n)) \subset O(fg)
\end{align*}
\]

Here \(f \cdot O(g(n)) = \{fh | h \in O(g(n))\}\).

**Addition:**

\[
\begin{align*}
\text{Action} & \quad f_1(n) \in O(g_1(n)), \ f_2(n) \in O(g_2(n)) \Rightarrow f_1f_2(n) \in O(|g_1| + |g_2|) \\
\text{Absorption} & \quad O(f_1(n)) + O(f_2(n)) = \begin{cases} O(f_1(n)), & \text{if } f_2 \in O(f_1) \\
                        O(f_2(n)), & \text{if } f_1 \in O(f_2) \end{cases} \\
\text{f > 0, g > 0} & \Rightarrow f + O(g(n)) \subset O(f + g)
\end{align*}
\]

Here \(f + O(g(n)) = \{f + h | h \in O(g(n))\}\).

**Scaling:**

\[
\begin{align*}
\text{Action} & \quad \forall k \in \mathbb{R}, f(n) \in O(g(n)) \Rightarrow kf(n) \in O(g(n)) \\
\text{Absorption} & \quad O(kf(n)) = O(f(n))
\end{align*}
\]

Some example subset relationships for various orders are \(O(1) \supset O(\log n) \supset O(n) \supset O(n^2) \supset O(n^k) \supset O(2^n) \supset O(n!) \supset O(n^n)\).
2.2. COMPLEXITY EXAMPLES

2.1.2 Comparing Orders

The set of order notation satisfies certain properties. For instance, all the five notations \((O, o, \Theta, \Omega, \omega)\) satisfy transitivity. This can be shown as:

**Transitivity:**
\[
f(n) \in \varphi(g(n)) \text{ and } g(n) \in \varphi(h(n)) \Rightarrow f(n) \in \varphi(h(n)), \forall \varphi \in \{O, o, \Theta, \Omega, \omega\}
\]

**Reflexivity:**
\[
f(n) \in \varphi(f(n)), \forall \varphi \in \{O, \Theta, \Omega\}
\]

**Symmetry:**
\[
f(n) \in \Theta(g(n)) \iff g(n) \in \Theta(f(n))
\]

**Transpose Symmetry:**
\[
f(n) \in O(g(n)) \iff g(n) \in \Omega(f(n))
\]
\[
f(n) \in o(g(n)) \iff g(n) \in \omega(f(n))
\]

2.2 Complexity Examples

In analyzing complexity for various algorithms, we can perform the analysis for the best case, the worst case, or the average case. In these examples we will be generally performing the worst case analysis. For instance if one is searching for a number in a list, the number could be the first one on the list where the search would stop at the first item in the best case scenario, at the last item in the worst case scenario, and in the middle item for the average case scenario.

2.2.1 \(O(1)\) Example

Take an algorithm that takes an input containing \(N\) numbers. The algorithm adds the first two numbers. The number of computations (additions) involved in the algorithm are independent of the problem size \(N\), and hence the algorithm has an order \(O(1)\). An example Python listing is shown in Program 2.1. Figure 2.1 shows a list where the algorithm adds the first two items, and hence has a complexity that is independent of the size of the list.
Program 2.1 $O(1)$ Example Code

```python
def AddFirst2(L):
    return L[0]+L[1]

L = [25,12,6,2,3]
result = AddFirst2(L)
print(result)
```

![Figure 2.1: $O(1)$ Example: Add First Two Items](image)

2.2.2 $O(\log(n))$ Example: Binary Search

An algorithm that has $O(\log_2(n))$ complexity is the binary search algorithm. The binary search algorithm that we will show here takes a list of numbers in an ascending order and also a number. If the number is in the list, the function returns the numerical index of the number in the list. If the number is not in the list, the function returns a zero.

The algorithm works as follows. It compares the number with the mid value of the list. If they are equal, then the algorithm returns the index, otherwise it applies the algorithm again to the upper list if the number was greater than the middle value, or else the lower list. Python listing for this code is shown in Program 2.2.

In the best case scenario, the algorithm can find the item in its first attempt. Every time, the comparison does not find the item, it deletes half of the list from the search. Hence, in the worst case, a list of size 8, gets reduced to the size of 4, then 2 and then 1. Hence, in the worst case, the complexity of the algorithm is of the order $O(\log_2(n))$. In the example shown in Figure 2.2, the algorithm goes through 3 searches on the list of size 8.

2.2.3 $O(n)$ Example: Linear Search

An algorithm that has $O(n)$ complexity is the simple linear search for a given number in a given list of unordered numbers.

The algorithm works as follows. It compares the number with the first value in the list. If they are equal, it returns the position of the number, one
2.2. COMPLEXITY EXAMPLES

Program 2.2 $O(\log n)$ Example Code: Binary Search

```python
def BinarySearch(L, a):
    size = len(L)
    if size <= 1:
        if a == L[0]:
            return 1
        else:
            return 0
    mid = size // 2
    if a == L[mid]:
        return mid+1
    if a < L[mid]:
        return BinarySearch(L[0:mid], a)
    if a > L[mid]:
        temp = BinarySearch(L[mid+1:size], a)
        if temp == 0:
            return 0
        else:
            return mid+1+temp
    return 0

L = [2, 4, 7, 12, 23, 41, 67, 79]
result = BinarySearch(L, 67)
print(result)
```

L = [2, 4, 7, 12, 23, 41, 67, 79]
result = BinarySearch(L, 67)
print(result)

**Figure 2.2:** Binary Search
in this case. If they are not equal it starts again on the rest of the list. If the
number is not found, it returns a zero. Python listing for this code is shown in
Program 2.3.

In the best case scenario, the algorithm can find the item in its first at-
ttempt. Every time, the comparison does not find the item, it moves to the
next one on the list. Hence, in the worst case, a list of size 8 gets searched 8
times. Hence, in the worst case, the complexity of the algorithm is of the or-
der $O(n)$. In the example shown in Figure 2.3, the algorithm goes through 3
searches on the list of size 8.

Program 2.3 $O(n)$ Example Code: Linear Search

```python
def LinearFind(L, a):
    size = len(L)
    if size == 0:
        return 0
    if a == L[0]:
        return 1
    else:
        temp = LinearFind(L[1:size], a)
        if temp == 0:
            return 0
        else:
            return 1 + temp
    return 0
```

$L = [12,41,7,2,67,4,79,23]$
$result = LinearFind(L,7)$
$print(result)$

2.2.4 $O(n^2)$ Example: Bubble Sort

An algorithm that has $O(n^2)$ complexity is the bubble sort. Bubble sort is the
algorithm used to sort an unordered list. The input to the algorithm is an
unsorted list of numbers, and the output is the list with the numbers sorted.

The algorithm works as follows. It compares numbers sequentially in pairs
till the lowest number is shifted all the way to the highest index. Then the
process is repeated to the list without the lowest number. Python listing for
this code is shown in Program 2.4.
2.2. COMPLEXITY EXAMPLES

(a) Linear Search on the List

(b) At Item 2

(c) At Item 3

Figure 2.3: Linear Search

Number of comparisons to obtain the lowest number will be \( n - 1 \), the number of comparisons to obtain the next lowest number will be \( n - 2 \), and so on. Hence the total number of comparisons will be

\[
\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \in O(n^2)
\]

2.2.5 \( O(n^p) \) Polynomial Complexity

We have seen examples of \( O(1) \), \( O(n) \), \( O(n^2) \), complexity. They are all specific examples of polynomial complexity of order \( O(n^p) \) for some whole number \( p \), specifically \( p = 0, 1, 2, \ldots \) respectively. We noticed that in the case of bubble sort, we had one loop nested in another and that gave us \( O(n^2) \) complexity. If we had three levels of nesting with each one iterating \( n \) times, we would obtain \( O(n^3) \) complexity. In general then, \( p \) number of loops, each one iterated \( n \) times gives us an \( O(n^p) \) complexity.
Program 2.4 \(O(n^2)\) Example Code: Bubble Sort

```python
def Bubblesort(L):
    size = len(L)
    for nlist in range(size - 1, 0, -1):
        for i in range(nlist):
            if L[i] < L[i + 1]:
                temp = L[i + 1]
                L[i + 1] = L[i]
                L[i] = temp

L = [25, 12, 6, 2, 3]
Bubblesort(L)
print(L)
```

2.2.6 \(O(2^n)\) Exponential Complexity

An algorithm to search for a numeric binary password with \(n\) digits has the worst case complexity that is of the order \(O(2^n)\). If there are three binary digits, then there are \(2^3\) different numbers possible. If all numbers are equally likely, then the algorithm will in the worst case have \(2^3\) comparisons. If there are \(k\) symbols for \(n\) digits, the number of comparisons are \(k^n\), and hence the complexity is exponential.

2.2.7 \(O(n!)\) Factorial Complexity

Let us assume that we have \(n\) different symbols or objects. If we place all the \(n\) symbols or objects in a straight ordered line one following the other (no repeats allowed), we will have \(n!\) ways to place them in that way. Now, if we have to guess the right pattern out of all these possibilities, this search task will have the complexity of order \(O(n!)\).

2.2.8 \(O(n^n)\) Complexity

Let us start with the same \(n\) different symbols or objects. If we place all the \(n\) symbols or objects in a straight ordered line one following the other as before except that this time repeats are allowed, we will have \(n^n\) ways to place them in that way. Now, if we have to guess the right pattern out of all these possibilities, this search task will have the complexity of order \(O(n^n)\).
2.3 Sorting Algorithms

We have already seen the bubble sort algorithm as a method to sort a list of numbers. In this section we present the summary of various sorting algorithms and their complexity order.

2.3.1 Bubble Sort

We have already presented this sorting technique when we showed how the sorting is accomplished by a pairwise comparison to bubble up the maximum value iteratively till the entire list is sorted as shown in the subsection 2.2.4. Please refer to Figure 2.4 for details of the application of this method to a given list. The complexity of the bubble sort is $O(n^2)$. 
2.3.2 Selection Sort

The selection sort is similar to the bubble sort, except that in the first round the location of the minimum number is found and then, the rightmost number and the minimum number exchange locations. Then the algorithm is repeatedly applied to get the next smallest number on the second rightmost location, etc. The complexity of the selection sort is $O(n^2)$.

**Program 2.5 Selection Sort**

```python
def Selection(L):
    size = len(L)
    for nlist in range(size-1, 0, -1):
        minV = L[0]
        index = 0
        for i in range(nlist + 1):
            if L[i] < minV:
                minV = L[i]
                index = i
        L[index] = L[nlist]
        L[nlist] = minV

L = [25, 12, 6, 2, 3]
Selection(L)
print(L)
```

Figure 2.5: Selection Sort
2.3.3 Insertion Sort

The insertion sort is also similar to the bubble sort. It starts with a list of one item, and then keeps inserting the next item at the proper place to keep the sorted list, which started with one item, growing in size till it covers the entire list. The complexity of the insertion sort is $O(n^2)$.

**Program 2.6 Insertion Sort**

```python
def Insertion(L):
    for nlist in range(len(L)-1):
        if L[nlist+1]>L[nlist]:
            continue
        else:
            for i in range(nlist+1):
                if L[nlist+1]<L[i]:
                    L.insert(i,L[nlist+1])
                    del L[nlist+2]
                    break

L = [25,12,6,2,3]
Insertion(L)
print(L)
```

Figure 2.6: Insertion Sort
2.3.4 Merge Sort

The merge sort is a recursive algorithm that follows a two step process of splitting the list into two halves, calling the merge sort on both halves, and then merging the results of the two calls. When the length of the list is one item, it returns the list. The complexity of the merge sort is $O(n \log n)$. The iterative splitting in half has $O(\log n)$ complexity as was the case with the binary search, and for each split, the merging process has a linear complexity. Hence, the total complexity has the order $O(n \log n)$.

Program 2.7 Merge Sort

```python
def Mergesort(L):
    lenL=len(L)
    if lenL <= 1:
        return L
    else:
        half=lenL//2
        L1=Mergesort(L[:half])
        L2=Mergesort(L[half:])
        if L1[half-1] <= L2[0]:
            L=L1+L2
        else:
            L=L2+L1
    return L

L = [25,12,6,2,3]
L = Mergesort(L)
print(L)
```

2.3.5 Quick Sort

The quick sort algorithm starts with picking a pivot element in the given array. Then we find the appropriate index for this pivot element so that members of the array that have values less than this pivot value are to one side (left or up for instance) of the array, and the values higher than the pivot value are on the other side. This pivot element does not move from this position any more. Now, the same algorithm is applied now to the left side and the right side recursively. The worst case complexity of the quick sort is $O(n^2)$ and the average and the best case is $O(n \log n)$. 
We can develop a simple algorithm where in each iteration we create three new lists. One list will contain all the members with values lower than the chosen pivot value, the second list containing values equal to the pivot value, and the third list containing values higher than that of the pivot. The lower and higher lists are reconfigured calling quick sort recursively, as shown in the program listing 2.8. We have chosen the first value to be the pivot value. Some other choice could also be made for picking the pivot value. For instance, we could randomize that choice. In our analysis shown in Figure 2.8 we have chosen 3 as the pivot in the first step of the algorithm.

The steps of the quick sort algorithm can be seen in the Program 2.8. If we take a list such as \( L = [5, 2, 6, 1, 12, 3, 25] \) and choose 5 as the pivot. This will create three lists. One list less than \( P = [2, 1, 3] \) will contain all the members having values less than the pivot, the list equal to \( P = [5] \) will contain as many copies of the pivot as there are in the original list, and the list greater than \( P = [6, 12, 25] \) will contain all the members whose values are greater than the pivot value. Then we call the algorithm again on the less than \( P \) and the greater than \( P \) lists to get them sorted. Finally, we concatenate the sorted version of less than \( P \) with equal to \( P \) with the sorted version of greater than \( P \).

Instead of creating new lists for every call of the function, we can also create an algorithm that performs the placement of the members of the list by in place shifting and exchanges. This is used in the program listing shown in 2.9.

The steps of the in-place quick sort algorithm can be seen in the Program 2.9. The program works by calling a partition function to find the location of the pivot element, and then calls the algorithm on the left and right sections of the list. The partition algorithm performs the sorting using an in-place algorithm.
Program 2.8 Quick Sort with New List Implementation

```python
def Quicksort(L):
    lessthanP=[]
équaltoP=[]
greaterthanP=[]
    if len(L) <= 1:
        return L
    else:
        Pivot=L[0]
        for n in L:
            if n < Pivot:
                lessthanP.append(n)
elif n > Pivot:
                greaterthanP.append(n)
else:
                equaltoP.append(n)

        return Quicksort(lessthanP)+equaltoP \ 
        +Quicksort(greaterthanP)

L = [3,2,6,12,25]
L = Quicksort(L)
print(L)
```

Figure 2.8: Quick Sort

2.4 Complexity Analysis Example

Given an algorithm we can attempt to find the cost of computation of that algorithm by studying the number of computations it takes and the cost of
2.4. COMPLEXITY ANALYSIS EXAMPLE

Program 2.9 Quick Sort with In-place List

```python
def QuicksortI(L, iFirst, iLast):
    if iFirst<iLast:
        q=Partition(L, iFirst, iLast)
        QuicksortI(L, iFirst, q-1)
        QuicksortI(L, q+1, iLast)

def Partition(L, iFirst, iLast):
    pivotvalue = L[iFirst]
    i = iFirst+1
    j = iLast
    while i<=j:
        while i <= j and L[i] <= pivotvalue:
            i = i + 1
        while i <= j and L[j] >= pivotvalue:
            j = j - 1
        if i<=j:
            temp = L[i]
            L[i] = L[j]
            L[j] = temp
        else:
            break
    temp = L[iFirst]
    L[iFirst] = L[j]
    L[j] = temp
    return j

def Quicksort(L):
    QuicksortI(L, 0, len(L)-1)

L = [3,2,6,12,25]
Quicksort(L)
print(L)
```
computations of various types involved in the algorithm. In general, the cost of comparison, multiplication, division, etc. will all be different. However, the main focus in algorithmic analysis is asymptotic and therefore, those differences become minor.

### 2.4.1 Median Computation Algorithm Analysis

In order to see what steps are involved in finding the complexity of an algorithm, let us study the code provided in Program 2.10.

**Program 2.10 Median Computation Code**

```python
def Median(L):
    while len(L) > 1:
        # n/2 times
        L.remove(max(L))
        # n/2 times
        if len(L) > 1:
            # n/2 times
            L.remove(min(L))
        else:
            return L[0]
    return L[0]
```

```plaintext
L = [3, 2, 6, 12, 25]
prompt (Median(L))
```

The comparison code to check the length is performed two times in the loop and overall each one is performed \( n/2 \) times. Hence the cost of the comparisons is \( n \) assuming unit cost for comparison. We will assume unit cost for each operation. In each loop, the max and min operations are performed one time. The cost of the max and min operation is equal to the size of the list at that iteration. As the size of the loop keeps decreasing at each iteration, the total cost of these two operations overall will be

\[
\sum_{i=1}^{n} i = \frac{n(n + 1)}{2} \in O(n^2)
\]

Hence, by combining the comparisons and the max and min operations, the total computation cost for the algorithm is obtained to be of the order \( T(n) = O(n) + O(n^2) = O(n^2) \), where \( T(n) \) represents the computational cost of the algorithm as a function of problem size \( n \).
2.4.2 Divide and Conquer Algorithm Analysis

Divide and conquer algorithms are typical of the recursion algorithms. Merge sort is an example of this type of an algorithm. The algorithm has a base case, when the length of the list is one, otherwise, it divides the problem into two pieces, obtains the result of the two and then combines the result to get the final solution. We will adapt this section of our book based on the material present in [LRSC01].

In a divide and conquer algorithm, there is a cost associated with the base case, which is applied once and in some instances more than once, but in any case a fixed number of times which is independent of \( n \), and then there is the recursive cost of dividing and combining.

If we denote by \( c_0 \) the number of times for the base case, and also assume that we get \( c_1 \) subproblems of size \( n/c_2 \), and also assume that the computational cost of dividing into subproblems is \( D(n) \) and that of combining the solutions of those subproblems is \( C(n) \), then the total cost of the computation is given by

\[
T(n) = \begin{cases} 
\Theta(1), & \text{if } n \leq c_0 \\
c_1 T\left(\frac{n}{c_2}\right) + C(n) + D(n), & \text{otherwise}
\end{cases}
\]  

(2.6)

For the merge sort algorithm, we have \( c_0 = 1, c_1 = c_2 = 2 \), the step to divide simply finds the middle element and hence takes fixed time and has complexity \( \Theta(1) \), the combine step merges the subarrays and hence has the complexity \( \Theta(n) \).

Combining these for the merge sort, we get

\[
T(n) = \begin{cases} 
\Theta(1), & \text{if } n = 1 \\
2T\left(\frac{n}{2}\right) + \Theta(n) + \Theta(1), & \text{otherwise}
\end{cases}
\]  

(2.7)

Specifically, after assuming a cost of \( t_1 \) time unit for a single computation, we get, for the merge sort the complexity iterative equation as

\[
T(n) = \begin{cases} 
t_1, & \text{if } n = 1 \\
2T\left(\frac{n}{2}\right) + t_1 n, & \text{otherwise}
\end{cases}
\]  

(2.8)

We can see from Figure 2.9 that shows the analysis for the merge sort algorithm that there are \( \log n \) levels and that each level has \( n \) computations.
CHAPTER 2. ALGORITHMS AND COMPLEXITY ANALYSIS

Hence, the total computations have the order $n \log n$. This method shown in Figure 2.9 is the recursion tree method to obtain a good guess for the complexity of the algorithm. For the rigorous estimation, we can use this guess to prove the actual complexity using the recursive computation equation as shown in Section 2.4.3.

2.4.3 Complexity Analysis using Recursive Computation Equation

A systematic way of finding the complexity order of an algorithm is to find the recursive equation for the computation. After that use a guess for the order and then try to prove by using the order inequality on the recursion equation.

The three steps involved in this analysis method are:

**Derive the Recursive Computation Equation**

This equation is derived from calculating how many computations are used in each step, and then how much is the problem size decreased before passing on to the function in the next recursive step.

**Guess the Order**

The next step is to guess the order of the complexity of the algorithm.
One method that can help in this process is to use the recursion tree. 

**Apply Asymptotic Induction**

Finally, in the last step we can use the inequality to prove the order of the complexity by using induction. Since the result has to be true asymptotically, the starting condition doesn't have to start at \( n = 1 \). It can start at any value of \( n \) which can be any finite positive integer.

As an example of this three step process, let us analyze the merge sort algorithm. Assuming a unit cost for a single step in equation 2.8, the recursive computation equation for merge sort is

\[
T(n) = \begin{cases} 
1, & \text{if } n = 1 \\
2T\left(\frac{n}{2}\right) + n, & \text{otherwise}
\end{cases} \tag{2.9}
\]

Strictly, the recurrence relationship, in fact is more accurately described by

\[
T(n) = 2T\left(\left\lfloor \frac{n}{2} \right\rfloor \right) + n \tag{2.10}
\]

The function \( \lfloor f \rfloor \) returns the greatest integer less than or equal to \( f \). Similarly, the function \( \lceil f \rceil \) returns the least integer greater than or equal to \( f \). We will ignore this difference, but it can be shown that it does not change the order of the algorithm. Based on the recursion tree analysis of the merge sort, we make a guess that its order is \( O(n \log n) \), i.e. \( \exists M > 0 \) such that \( \exists N > 0, N \in \mathbb{N} \) so that \( \forall n > N, |f(n)| \leq M|n \log n| \). This shows that the inequality is satisfied asymptotically, i.e.

\[
\exists M, \lim_{n \to \infty} T(n) < Mn \log n \tag{2.11}
\]

To apply induction, we have to show that
1. the statement is true for the base case, and
2. given, that it’s true for all lower values of \( n \), then it must be true for the next higher value

To prove the statement for the base case is easy. Now, the inequality is not true for \( n = 1 \) because \( n \log n = 0 \) for that case. However, for \( n = 1 \) we can prove the relationship. Notice that if we have the inequality true for some \( M \), then it must be true for all \( K > M \). Hence, we can choose a high enough value of the constant so that the inequality is true for \( n = 1 \). Without a loss of generality, we assume that our constant satisfies the inequality for \( n = 1 \).
For the second step, we now assume that the inequality is true for $\lfloor n/2 \rfloor$, and now we attempt to prove it for $n$. We assume that the function $T(n)$ is a non decreasing function of $n$. We obtain

$$T(n) = 2T(\lfloor n/2 \rfloor) + n \leq 2T(\frac{n}{2}) + n \leq 2M\frac{n}{2}\log\frac{n}{2} + n = Mn\log n - M\log 2 + n \leq Mn\log n$$ (2.12)

In fact, there is a general theorem as presented in [LRSC01], that we will reproduce here, which can help in deriving the complexity of an algorithm based on its recursive computation equation. For the proof and discussion, please refer to [LRSC01], where this theorem is called the master theorem.

**Theorem 2.9** (Main Theorem). Given $T(n) = aT(n/b) + f(n)$, $\forall n \in \mathbb{N}, n > 0$, and where $a > 0, b > 0$, and $f(n)$ is asymptotically positive then

1. If $\exists \epsilon > 0 \text{ s.t. } f(n) = \Theta(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$,
2. $f(n) = O(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a \log n})$, and
3. if $\exists \epsilon > 0 \text{ s.t. } f(n) = O(n^{\log_b a + \epsilon})$ and $\exists c < 1$, s.t. $af(n/b) < cf(n)$, then $T(n) = \Theta(f(n))$.

In this theorem $n/b$ can be replaced by $\lceil n/b \rceil$ or $\lfloor n/b \rfloor$, and the results would still be valid.

### 2.4.4 Average Case Analysis: Randomizing Algorithms

When the complexity analysis is performed for an algorithm, the aim is to find the complexity for the best case, the worst case and the average or the mean case. After studying how many computations we have to perform we derive the recursive computation equation for $T(n)$ for the best case and the worst case. To compute the average case, we have to look at all possibilities of the inputs to the algorithm. We use probability distribution over the inputs and then compute the expected value of $T(n)$, since in this case, for a fixed $n$, $T(n)$ will be random variable.

We consider $T(\omega, n) : \Omega \times \mathbb{N} \rightarrow \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers. This makes $T(\omega, n)$ a stochastic process, where the probability space is given by the triplet $(\Omega, \mathcal{F}, P)$. Probability space (the triplet) is a measure space from
measure theory, in which the measure is normalized. \( \Omega \) is the set of all outcomes, \( \mathcal{G} \) is the set of events which is a sigma algebra.

**Definition 2.10. Sigma algebra** \( \mathcal{G} \) is a collection of the subsets of \( \Omega \) that satisfies the following properties:

1. \( \Omega \in \mathcal{G} \),
2. If \( \forall i \in \mathcal{A}, \) where \( \mathcal{A} \subset \mathbb{N}, U_i \in \mathcal{G}, \) then \( \bigcup_{i \in \mathcal{A}} U_i \in \mathcal{G}, \) and
3. \( \mathcal{B} \in \mathcal{G} \Rightarrow \mathcal{B}^c \in \mathcal{G} \)

This essentially implies that the universal set, which is also called the fundamental probability set or the sure event is in the sigma algebra (also called sigma field), countable union of members of the sigma algebra is in the sigma algebra, and complement of a member of a sigma algebra also belongs to the same sigma algebra. These three, because of de Morgan’s law imply that countable intersections of members of a sigma algebra also belong to the same sigma algebra. In measure theory ([RFH10]), the members of the sigma algebra are called measurable sets, and in probability theory, those are called events. An excellent introduction to probability theory, random variables, and stochastic processes is presented in [Tuc14].

**Definition 2.11.** The **probability function** \( P : \mathcal{G} \rightarrow [0, 1] \) is a measure on \( \mathcal{G} \), which is normalized to one. A non-negative countably additive function on a sigma algebra whose value is zero for the empty set is called a measure. Hence, the probability function satisfies the following properties:

**Measure:**

1. \( P(\emptyset) = 0, \) where \( \emptyset \) is the empty set,
2. \( P\left(\bigcup_{i=1}^{\infty} U_i\right) = \sum_{i=1}^{\infty} P(U_i), \) where \( \forall i, U_i \in \mathcal{G}, \) \( U_j \cap U_k = \emptyset, j \neq k \)

**Normalized:**

1. \( P(\Omega) = 1 \)

The countable additivity axiom implies that the probability of a countable set of disjoint events is the countable sum of the probabilities of those events. It is important to note that since the right hand side is a series, the countable sum implies the limit of the sequence of the partial sums.

Random variables are measurable functions from probability space to some other measure space. We will consider the random variables to the space \( \mathbb{R}, \)
with its usual topology and the Borel measure. The Borel measure space is the triplet \((\mathbb{R}, \mathcal{B}, \mathcal{L})\), where \(\mathcal{B}\) is the Borel field, which is the smallest sigma algebra of subsets of subsets of \(\mathbb{R}\) that contains open intervals, and finally \(\mathcal{L}\) is the Lebesgue measure, which is the unique extension of length of intervals to the Borel field using the Caratheodory construction (see [RFH10] for details).

**Definition 2.12.** A random variable \(X(\omega) : \Omega \to \mathbb{R}\) is a function, such that

\[
\forall b \in \mathcal{B}, X^{-1}(b) \in \mathcal{S} \quad (2.13)
\]

This essentially allows us to calculate probabilities of the random variable taking values on the real number line. The definition 2.12 tells us that we can calculate the probability that the random variable would take values between two real numbers \(a\) and \(b\). To obtain this probability, we use the probability function on the Borel field of events in the co-domain of the random variable. Hence, we compute

\[
\text{Probability (}X \in (a,b)\text{)} = P (\{\omega | X(\omega) \in (a,b)\}) \quad (2.14)
\]

With an abuse of notation, this probability is generally shown to be \(P [X \in (a,b)]\). Similarly, the probability of any Borel subset of the set \(\mathbb{R}\) is computed. It is clear that for every random variable, by its very definition, for any \(x \in \mathbb{R}\), we can compute the probability \(P [X \in (\infty,x)]\). This will be a number between 0 and 1 since \(P [X \in (\infty,x)] \in [0,1]\). Moreover, if \(x \in \mathbb{R}\), \(y \in \mathbb{R}\), and \(y > x\), then \((\infty,x) \subset (\infty,y)\), and due to the monotonicity of probability measure, we can see that \(P [X \in (\infty,x)] \leq P [X \in (\infty,y)]\). Using this we can obtain a non-decreasing function whose values are between 0 and 1 and is called a distribution of the random variable \(X\).

**Definition 2.13.** The probability distribution function \(F_X : \mathbb{R} \to [0,1]\) of a random variable \(X\) is defined as \(F_X(x) = P[X \leq x]\). □

Every random variable has a distribution, but it might not have a probability density function whose integral on \((\infty,x)\) would give the distribution function. However a class of distribution functions have a guaranteed probability density function.
2.4. COMPLEXITY ANALYSIS EXAMPLE

**Definition 2.14.** If the distribution function $F_X : \mathbb{R} \rightarrow [0, 1]$ of a random variable $X$ is **absolutely continuous**, then $\exists f_x \in L_1(\mathbb{R})$, we call the **probability density function** of $X$, such that

$$F_x(x) = \int_{-\infty}^{x} f_x(t) \, dt$$

(2.15)

Now, if we only at most countable number of events possible, then the random variable is called discrete. It's distribution function is a piecewise constant function, and its density function, in fact, is an impulse train, impulse being the Dirac delta distribution (a functional in a purely vector space method terminology). Let us take a discrete random variable that has non-zero probability only at discrete points on the real line at $\{x_1, x_2, \ldots\}$. Let the probabilities at those points be given by the probability function as $P(x_1), P(x_2), \ldots$. Then its distribution function, because of its discrete measure, would be given by

$$F_x(x) = \sum_{x_i}^{x} P(x_i) \quad \forall x_i \in (-\infty, x]$$

(2.16)

If we compare two random variables defined on the same probability space, one common way to claim that one variable is greater than the other one if its expected value is higher than the other. For instance, if we compare average number of messages received in a day for two different entities. For each entity, daily average number of messages is a random variable. Some days one entity received might have received more than the other, but some other days, the reverse might be true. However, we should still be able to claim that in some respect one is greater than the other. For this, we define a linear operator called the expectation of random variables that gives a real number for integrable random variables in a specific sense defined below.

**Definition 2.15.** The expectation $E(X)$ of a random variable maps the set of random variables to $[0, 1]$ and is the Lebesgue integral of the random variable over its probability measure, i.e. $E(X) = \int X \, dP$.

(2.17)

We say that a random variable has a mean and show it as $X \in L_1(\Omega, \mathcal{G}, P)$, if the Lebesgue-Stieltjes integral of the random variable with respect to its distribution function over $\mathbb{R}$ exists and is finite. This is equal to its expectation. In general, the expectation of a function $g(X)$ of a random variable is given by:

$$E(g(X)) = \int g(x) \, dF_X(x)$$
In terms of the probability density function, this integral is given by

\[ E(g(X)) = \int g(x) f_x(x) dx \quad (2.18) \]

If the random variable is discrete, then these integrals turn out to be in fact summations. The mean, in that case for the discrete random variable would be

\[ E(g(X)) = \sum g(x_i) P(x_i) \quad (2.19) \]

Now, let us go back to studying the computational cost, \( T(\omega, n) : \Omega \times \mathbb{N} \to \mathbb{N} \) as a stochastic process. For a fixed \( n \), \( T(\omega, n) \) becomes a random variable. For instance, \( (T(\omega, n), T(\omega, n), \cdots) \) is a sequence of random variables. By applying the expectation operator on each of these random variables, we obtain, \( (E(T(\omega, n)), E(T(\omega, n)), \cdots) \), a sequence of real numbers. This sequence of real numbers is a function of \( n \), and it is the asymptotic properties of this function, that average complexity analysis is involved with.

### 2.4.5 Example of an Average Case Analysis

Now we will apply the analysis on a simple specific algorithm to be able to compare the complexity of the best, worst, and the average case. In order to make the analysis more structured, we will consider the computation machine to be a Turing machine, since Turing machines are universal computers. We will follow the example algorithm from [Kem85].

**Turing Machine Description for the Example**

A Turing machine’s tape has the initial data which is an \( n \)-digit binary number, as shown in Figure 2.10. The location of the read/write head is shown initially and also at the final state. The output should also be a binary number but with \( n + 1 \) digits \( b_i \) which should be obtained by adding a 1 to the input binary number. The digits should satisfy the following relationship.

\[ n \sum_{i=1}^{n} b_i 2^i = 1 + \sum_{i=1}^{n-1} a_i 2^i \quad (2.20) \]

The algorithm to accomplish this task by the Turing machine is shown in Figure 2.11.
Let us study Figure 2.12 to understand the algorithm. We just need to add 1 to the binary number on the Turing machine input/output tape. Hence, in Figure 2.12, the only time augend is a 1 is when the addend digit is \( a_0 \). For all other \( a_k \) where \( k \neq 0 \), augend is 0. So, when \( a_0 \) is being added to 1, the carry is 0.

**Case** \( a_0 = 0 \) In the case, looking at Figure 2.12, when \( a_0 \) is 0, then the new digit \( b_0 \) becomes 1, but we also get \( b_k = a_k \) for the rest. So the Turing machine
needs to leave all the other digits untouched. Let us study the Figure 2.11 to see how the Turing machine transitions accomplish this task. The initial state of the machine is $q_0$ when the input symbol in this case is 0. The label on the arc showing the transition from state $q_0$ to $q_1$ shows that the read/write head of the machine changes the symbol 0 to 1 on the tape and the read/write head shifts to the right after making this change. Now, since $a_0$ was the first digit, we have a * to the right of that in the tape. According to the transition diagram in Figure 2.11, we see that when we are in state $q_1$ and the input is a □, the Turing machine leaves the □ as is and moves to the left reaching the final state $q_2$ and stopping. Now, the machine is in the final state as shown in Figure 2.10.

**Case $a_0 = 1$** In the case, looking at Figure 2.12, when $a_0$ is 1, then $a_0$ needs to change to 0 and there’s a carry of 1 to decide what would happen to $a_1$ now. We can see, corresponding to this, in Figure 2.10 that when the machine is in state $q_0$ and the input is 1, then, as the loop on $q_1$ shows, we stay in the same state, change the input symbol at 0 which is $a_0$ to 1 which is $b_0$. If the machine keeps getting 1 now, it keeps changing it to 0 and moving to the left. This is also clear from analyzing these additions from Figure 2.12. If at any step, the machine encounters a 0, it goes to state $q_1$, changes the 0 to 1 on the tape. On the other hand if it never encounters a 0, then it will encounter a * after $n$ steps, and according to the transition diagram, it will change that to a 1, making $b_n$ equal to 1, and then move to the right on the tape, and changing the state to $q_1$. In state $q_1$, irrespective of reading a 0 or a 1 you keep moving right without changing the symbols till you hit a □, the right end of the tape, and step one step to the left to finish there.

**Complexity Analysis of the Example**

For the complexity analysis, let us study the three cases.

**The Best Case Analysis** The best case is the one where $a_0$ is equal to 0. In that case the Turing machine changes the value to 1, moves to the right, reads a □, and moves to the left. Hence, it involves two moves, which is independent of the size of the problem that is $n$. Hence, the complexity is of order $O(1)$. From a probabilistic point of view, the probability of getting a 0 or a 1 at any digit is liking getting a tail or a head on tossing of an un-biased coin. Hence, the probability of getting a 0 at the first digit is $2^{-1}$.

**The Worst Case Analysis** The worst case is the one where $a_k = 1$ for all $k$ upto $n$. In that case the Turing machine changes the value to 0, moves to the
left repeatedly, till it hits the leftmost $\Box$, and then starts moving to the right all the way till it hits the rightmost $\Box$ and then makes one additional left move to reach the final state. Hence, it involves $2n + 2$ moves, which is linear with respect to the size of the problem that is $n$. Hence, the complexity is of order $O(n)$. From a probabilistic point of view, the probability of getting all 1 is $2^{-n}$.

The Average Case Analysis To estimate the average case we need to study the asymptotic properties of the expectation of the probabilistic computation, i.e. we are looking for the asymptotic properties of

$$E(T(n)) = \sum_{i=1}^{k(n)} p(i)T_i(n)$$

(2.21)

where, $p(i)$ is the probability of obtaining an input corresponding to $i$, one of the possibilities when the problem size is $n$. Now, since each digit can have two value independently of all other digits, the total number of choices are $2^n$, and we can assume that each one is equally probable. Hence, we have that, in Equation 2.21, $k(n) = 2^n$.

Now, the set of all possible numbers can be divided into the following collection of sets. Let $U_m$ be the set of binary numbers having $m$ number of 1s from the right. Now the set of all $n$ digit numbers is a union of these sets, i.e.

$$X = \bigcup_{m=0}^{n} U_m$$

(2.22)

The probability of each of these sets is $2^{-m}$. The number of computations required on each of these is of the order $O(m)$. Hence, we need to compute

$$E(T(n)) = \sum_{0}^{n} m2^{-m}$$

(2.23)

We know that, a sum of the geometric series for $|x| < 1$,

$$\sum_{k=1}^{\infty} x^k = \frac{1}{1-x}$$

(2.24)

By differentiating and both sides and rearranging, we get,

$$\sum_{k=1}^{\infty} kx^k = \frac{x}{(1-x)^2}$$

(2.25)
Applying this in Equation 2.23, we get

\[ \lim_{n \to \infty} E(T(n)) = \sum_{m=0}^{n} m2^{-m} = O(1) \]  \hspace{1cm} (2.26)

\section*{2.5 Generating Functions Method to Solve Recursive Equations}

\section*{2.6 Average Case Analysis of Algorithms}

\section*{2.7 Bibliographical Notes}

We discuss bibliography related to this section …

\section*{2.8 Exercises}

\textbf{Problem 2.1.} Give an example of two functions \( f(n) \) and \( g(n) \) such that simultaneously, \( f(n) \neq O(g(n)) \) and \( f(n) \neq \Theta(g(n)) \).
This chapter is devoted to presenting the fundamentals of language theory, grammars for languages, and machines that interact with languages. These three present extremely important aspects of the subject that leads to the complexity theory of algorithms, which is very important in the study of complex networks.

### 3.1 Languages, Grammars, and Machines

This section will introduce languages, grammar, and machines by presenting one example of each, such that the three examples are related to each other.

#### 3.1.1 Language

Language is a set of finite length strings obtained using symbols from a finite alphabet. For instance, let us consider the alphabet $\mathcal{A} = \{a, b\}$ consisting of the finite number of symbols $a$ and $b$. A language over $\mathcal{A}$ is just one set of strings made up of symbols from $\mathcal{A}$. A language can be finite or infinite depending on how many members it has (cardinality).

**Example 3.1.** For our example in this section let us consider the following language.
\[ \mathcal{L} = \{ a^n b : n \in \mathbb{N} \} \] (3.1)

The language \( \mathcal{L} \) can be enumerated as \( \mathcal{L} = \{ ab, aab, aaab, \ldots \} \).

We define concatenation (shown as \( \otimes \)) of two strings as creating a new string by joining the two together. For instance if we have \( s_1 = abb \), and \( s_2 = baab \), then \( s_1 \otimes s_2 = abbaab \). It is clear that \( (s_1 \otimes s_2) \otimes s_3 = s_1 \otimes (s_2 \otimes s_3) \), and also that \( s_1 \otimes s_2 \neq s_2 \otimes s_1 \). These show that the concatenation operation is associative but not commutative.

Consider two alphabet sets given \( \mathcal{A} \) and \( \mathcal{B} \). For instance, let us take \( \mathcal{A} = \{ a, b \} \) and \( \mathcal{B} = \{ c, d \} \), then we define concatenation of sets \( \mathcal{A} \) and \( \mathcal{B} \) to be \( \mathcal{A} \mathcal{B} = \{ x \otimes y : x \in \mathcal{A}, y \in \mathcal{B} \} \). For the example we are considering we get \( \mathcal{A} \mathcal{B} = \{ ac, ad, bc, bd \} \). Using the concatenation of sets, we also define powers of sets.

Using the same example, we define \( \mathcal{A}^0 = \{ \lambda \} \), \( \lambda \) being the null string, that has no symbols. \( \lambda \) has the property that for any string \( s \), we have \( s \otimes \lambda = \lambda \otimes s = s \). For the non-zero powers we have, \( \mathcal{A}^1 = \mathcal{A} \), \( \mathcal{A}^2 = \mathcal{A} \mathcal{A} \), etc. We also define the Kleene star of \( \mathcal{A} \), denoted by \( \mathcal{A}^* \) as

\[ \mathcal{A}^* = \bigcup_{k=0}^{\infty} \mathcal{A}^k \] (3.2)

Kleene star of \( \mathcal{A} \) is essentially the set of all finite strings including \( \lambda \) that can be created from the symbols in \( \mathcal{A} \). Hence, for \( \mathcal{A} = \{ a, b \} \), we have \( \mathcal{A}^* = \{ \lambda, a, b, aa, ab, bs, bb, aaa, \ldots \} \). We can also define \( \mathcal{A}^+ = \{ a, b, aa, ab, bs, bb, aaa, \ldots \} \), which does not contain the null string.

For a given alphabet \( \mathcal{A} \), \( \mathcal{A}^* \) is the set of all finite strings, and any subset of \( \mathcal{A}^* \) including the empty set \( \phi \) is a language over \( \mathcal{A} \). Hence any language \( \mathcal{L} \) over \( \mathcal{A} \) satisfies \( \mathcal{L} \in 2^{\mathcal{A}} \). Notice that \( \phi \) and \( \{ \lambda \} \) are two different languages, the first one being the empty set, and the second one being a language containing only the null symbol. Since, a language \( \mathcal{L} \) is a set we can do set operations with it. For instance \( \mathcal{L}^c \) is the complement of the language \( \mathcal{L} \). Similarly, we can take union and intersection of languages, and also concatenation as discussed above.

A semigroup is a set with an associative binary structure. More formally,

**Definition 3.2.** A semigroup \( (S, \otimes) \) is a set \( S \) with a binary operation \( \otimes \) such that:

1. **Closure:** \( \forall x, y \in S, x \otimes y \in S \)
2. **Associativity:** \( \forall x, y, z \in S, (x \otimes y) \otimes z = x \otimes (y \otimes z) \)
A monoid is a semigroup with an identity. Formally,

**Definition 3.3.** A monoid \((M, \otimes)\) is a set \(M\) with a binary operation \(\otimes\) such that:

1. **Closure:** \(\forall x, y \in M, x \otimes y \in M\)
2. **Associativity:** \(\forall x, y, z \in M, (x \otimes y) \otimes z = x \otimes (y \otimes z)\)
3. **Identity:** \(\exists I \in M, \forall x \in M, x \otimes I = I \otimes x = x\)

This shows that any language over a given alphabet \(A\) is a semigroup, and if the language contains \(\lambda\), then it is a monoid.

### 3.1.2 Grammar

To understand what a grammar is and how it works, we will just take a look at an example shown below.

**Example 3.4.** Consider the grammar shown in the Table 3.1.

<table>
<thead>
<tr>
<th>Substitution</th>
<th>Rule Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\langle S \rangle ) ::= (a \langle A \rangle)</td>
<td>1</td>
</tr>
<tr>
<td>(\langle A \rangle ) ::= (a \langle A \rangle \mid b)</td>
<td>2a</td>
</tr>
</tbody>
</table>

This grammar will generate a language from the alphabet \(\{a, b\}\), which are the terminal symbols in the rules of the grammar shown above. The meaning of the above two rules is the following. Starting with the starting symbol \(S\), we can replace it by a string \(aA\), where \(A\) is also a non-terminal symbol. Using the notation used in [Lin11], we can show the applied rule as \(S \rightarrow aA\) which is the rule 1. The grammar shown above is using a formatting that shows terminal symbols (which become the symbols of the alphabet) in bold case, and non-terminals in upper case and surrounded by angled brackets \(\langle \cdot \rangle\). One application of this rule gives the derivation

\[ S \Rightarrow aA \quad (3.3) \]

Now, \(A\) is not a terminal symbol. Therefore, we can not stop the derivation here. The other rule indicates that \(A\) can be replaced either by \(aA\) or \(b\). This
rule is in fact two rules, \( A \to aA \) which we call rule 2\( a \) and rule 2\( b \) which is \( A \to b \). So, for instance, if we apply rule 2\( b \) to \( A \) in the string \( aA \), we get \( ab \) and that is the final string since there are no more non-terminals in this string. So, we have shown that

\[
S \Rightarrow ab
\]  

(3.4)

where the symbol \( \Rightarrow \) means that the right hand side the string is derivable from the left hand side one. This two step derivation is shown next.

\[
S \overset{1}{\Rightarrow} aA \overset{2b}{\Rightarrow} ab
\]  

(3.5)

In this notation, the string on top of the symbol \( \Rightarrow \) shows the rule applied in that step. We can generate many other strings as follows.

\[
S \overset{1}{\Rightarrow} aA \overset{2a}{\Rightarrow} aaA \overset{2b}{\Rightarrow} aab, \quad S \overset{1}{\Rightarrow} aA \overset{2a}{\Rightarrow} aaA \overset{2a}{\Rightarrow} aaaA \overset{2b}{\Rightarrow} aaab
\]  

(3.6)

It is clear that the collection of all strings that this grammar can generate is clearly the language of Example 3.1, since this grammar can generate any string of type \( a^n b \) by applying rule 1, followed by \( n - 1 \) applications of rule 2\( a \) and then one final application of rule 2\( b \) giving us \( S \Rightarrow a^n b \).

Motivated by this example, we can give a formal definition of grammars. We noticed in this example that there were two non-terminal symbols \( S \) and \( V \), two terminal symbols \( a \) and \( b \), a special non-terminal symbol \( S \) which was the starting non-terminal symbol, and then we have three production rules, 1, 2\( a \), and 2\( b \). Hence, formally, we have the following.

**Definition 3.5. (Grammar):** A grammar for a language is a quadruple \((N, T, S, \mathcal{P})\), where

1. **Variables:** \( N \) is a finite set of non-terminal symbols
2. **Terminals:** \( T \) is a finite set of terminal symbols
3. **Start Variable:** \( S \in V \) is the starting variable
4. **Productions:** \( \mathcal{P} \) is a finite set of production rules

The set of all strings of terminal symbols produced by sequentially applying the grammar rules starting with the start variable, and then replacing all the variables by choosing any applicable production rules is the language corresponding to that grammar.
3.1.3 Machine

We will study a finite automata as an example of a machine and show how it relates to the language of Example 3.1, and the grammar of Example 3.4.

Example 3.6. Consider the automata shown in Figure 3.1.

\[ \text{Figure 3.1: Finite Automata Example} \]

The automata starts in state $q_0$ indicated by the left arrow. Then it can accept a single $a$ to move to state $q_1$, and then can accept any number of $a$ symbols and then one more $b$ to reach the final state and get terminated. Any other sequence of symbols will not terminate the automata in the final state $q_2$. When the automata is in state $q_2$ and any new symbol $a$ or $b$ is encountered, the system goes to state $q_3$ from which no string can be accepted. The symbol $a, b$ on the arc means that if any of those two symbols are encountered, the state transitions from $q_2$ to $q_3$. When the automata terminates, whatever sequence of symbols it had encountered, is the string it accepts. The set of all strings it accepts is the language of the automata.

For this automata to accept the string $ab$ it goes through two transitions. First one from $q_0$ to $q_1$ and the second one from $q_1$ to $q_2$. These transitions can be shown as

\[ q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_2 \] (3.7)

In this notation, the string on top of the symbol $\Rightarrow$ shows the symbol used for making the transition from the state on the left to the state on the right of the symbol. We can generate many other strings as follows.
3.2 Finite State Automata, Regular Languages, and Grammars

Languages can be categorized based on certain properties they have. Languages that are accepted by Deterministic Finite State Automata (DFA) are called regular languages. Grammars that generate regular languages are called regular grammars. These grammars have a specific structure and they are either right-linear or left-linear grammars. These languages, and their corresponding grammar and automata are explained next.

3.2.1 Finite State Automata

As we saw in Example 3.6, and Figure 3.1, a finite automata has an initial state the system starts in, and then arcs coming out of each state shows what symbol, shown as the directed arc label, makes the system transition from that state to another state. If a string of symbols is given, and at the application of the last symbol, the system reaches a final state (shown with double lined circle), then the automata accepts that string, otherwise it rejects that string. Hence, the automata has a finite set of internal states $Q = \{q_0, q_1, q_2\}$ for this example with $q_0$ being the initial state, and $q_2$ being the final state, a finite set of input alphabet, $\Sigma = \{a, b\}$, and a transition function $\delta$ that takes two inputs, the current state and the input symbol to compute the next state to transition to, for example $\delta(q_0, a) = q_1$. We present this definition formally as

**Definition 3.7. Deterministic Finite Automata (DFA):** A deterministic finite automata is a quintuple $(Q, \Sigma, \delta, q_0, F)$, where

1. **States:** $Q$ is a finite set of states
2. **Alphabet:** $\Sigma$ is a finite set of symbols, called the alphabet
3. **Transition Function:** $\delta : Q \times \Sigma \rightarrow Q$ is the transition function
4. **Initial State:** $q_0$ is a initial state
5. **Final Set of States**: $F$ is the set of final or terminal states

We will now take a slightly different view of the representation of the dynamics of finite automata. We have represented finite automata using graphs where we have shown the input transition relationship on arcs. Consider the deterministic finite automaton shown in Figure 3.2 which accepts the language $(ab)^n$. Compare this representation to Figure 3.2, where we see an input string on an input tape, and the automaton having a read-head which points to the symbol the machine reads. The transition step in the Figure 3.2 based on the transition dynamics shown in Figure 3.2 are as follows. The read-head starts at the left slot of the tape. The state of the automaton is shown on the read-head itself, and it is $q_0$ as shown in Figure 3.2. Now, the read-head reads the input, and transitions its state, and then shifts right. In this case, it reads the symbol $a$ and changes its state to $q_1$, and then shifts right. It does this step repeatedly following its transition rules till it reaches the end of the tape, at which point, if it finds itself in a final state, it accepts the string, otherwise it rejects it. Notice that after reading the last symbol it does not shift right, since there is no right after the last symbol.

![Figure 3.2: DFA: $(ab)^n$ Acceptor](image)

There can be many different DFAs that are essentially equivalent in their language acceptance. A DFA that has the minimum number of states can be constructed by using the concept of right invariant equivalence relation. **Myhill-Nerode theorem**, that shows this relationship is used to establishing the minimal DFA (see [Sud05] for details).

The automata, we have been studying are called deterministic because the transition function is a *total* function. That means that the domain of the
transition is the entire product space $Q \times \Sigma$ and the transition is a function, i.e. for any state and input, there is a unique transition to a state in $\Sigma$.

Nondeterministic automata differ from deterministic ones in three different ways.

1. They can have more than one transition from a given state and a given input. Hence their transition map is $\delta : ((Q \cup \lambda) \times \Sigma) \rightarrow 2^Q$ rather than $\delta : (Q \times \Sigma) \rightarrow Q$ which was the case for deterministic ones. This means that the transition function maps to a set of states rather than a state. A set of states is a member of the power set $2^Q$.

2. They can transition without any input symbol as shown in the domain of the transition function $\delta : ((Q \cup \lambda) \times \Sigma) \rightarrow 2^Q$.

3. Their transition function doesn’t have to complete, i.e. they might not show all the transitions. Which means the transitions that are not shown do not lead to the final set of states.

As an example of a Non-deterministic Finite Automata (NFA), consider the Figure 3.4, in Example 3.8.

**Example 3.8.** Consider the automata shown in Figure 3.4.

We can see that from state $q_1$ with input $a$ there are two transitions, one to $q_1$ and another to $q_2$. Hence $\delta(q_1, a) = \{q_1, q_2\}$. We can also see that from state $q_0$, the system can transition automatically, without any input, to the final state $q_2$. We also see that from $q_0$, a transition with an input $a$ is not specified. Let us ascertain what strings starting from the state $q_0$ will be accepted by this automata. The string containing the empty symbol $\lambda$ going from $q_0$ directly to $q_2$ is accepted, strings of type $ba^n$ for $n > 0$ are accepted, the string $bb$ is accepted, and finally strings $ba^n b$ are also accepted. In summary the
The language accepted by this nondeterministic automata is \( \{ \lambda, ba^n b, ba^k \}, n \geq 0, k > 0 \).

The languages that DFAs and NFAs accept are the same (for proofs and details please refer to [Sud05], [Sip06], [HMU07], [Lin11], and [DDQ78]). Every deterministic automata is automatically a nondeterministic one. In the other direction, for every nondeterministic automata we can design a deterministic automata that accepts the exact same language. The idea behind this conversion is to think in terms of more states the system can be in, which is encapsulated in these new states.

We present the definition of the nondeterministic finite automata formally as

**Definition 3.9. Nondeterministic Finite Automata (NFA):** A nondeterministic finite automata is a quintuple \((Q, \Sigma, \delta, q_0, F)\), where

1. **States:** \(Q\) is a finite set of states
2. **Alphabet:** \(\Sigma\) is a finite set of symbols, called the alphabet
3. **Transition Function:** \(\delta : ((Q \cup \lambda) \times \Sigma) \rightarrow 2^Q\) is the transition function
4. **Initial State:** \(q_0\) is an initial state
5. **Final Set of States:** \(F\) is the set of final or terminal states

---

### 3.2.2 Finite State Automata as Transducers

There are two ways that finite state automata have been used as transducers of strings, i.e., given an input string, they produce an output string based on a transformation that they perform due to their structure. These two types are called Mealy machines and Moore machines which are described next.
Mealy Machine

A Mealy machine is a finite state automata transducer for which the output is based on transition. The output symbol is based on the current state and the input symbol.

Example 3.10. Consider the automata shown in Figure 3.5.

![Figure 3.5: Mealy Machine](image)

What we notice about this diagram is that each arc has a label with two symbols of the type $a/b$. The symbol $a$ belongs to the input alphabet, and symbol $b$ belongs to the output alphabet. The Figure shows that when the machine is in state $q_0$ and an input of 0 is asserted, then an output of $e$ is obtained. In other words the output function is $y(q_0, 0) = e$. Similarly, when the state is $q_1$ and the input is 0, the output is $o$, i.e. $y(q_1, 0) = o$. There is no final state. The machine operates till the last symbol of the input string is encountered. Let us take an arbitrary input string and ascertain its output from this machine. If we take 01101 as the input, we start in state $q_0$, for the first 0, we get $y(q_0, 0) = e$ and we stay in $q_0$. Next we get 1 as the input, and the output for that is $y(q_0, 1) = o$, and we transition to state $q_1$. Following similarly, the output string we get is $eooeo$. The machine is a parity checker for the number of 1s encountered from the beginning to the current time. The output $e$ shows even parity, and the output $o$ shows odd. We can show this machine transforming input string to output string as:

$$01101 \xrightarrow{M} eooeo$$

Formally, the Mealy machine is given by:

**Definition 3.11. Mealy Machine**: A Mealy machine is a sextuple $(Q, \Sigma, \Gamma, \delta, y, q_0)$, where

1. **States**: $Q$ is a finite set of states
3.2. **FINITE STATE AUTOMATA, REGULAR LANGUAGES, AND GRAMMARS**

2. **Input Alphabet**: $\Sigma$ is a finite set of input symbols, called the input alphabet
3. **Output Alphabet**: $\Gamma$ is a finite set of output symbols, called the output alphabet
4. **Transition Function**: $\delta : (Q \times \Sigma) \rightarrow Q$ is the state transition function
5. **Output Function**: $y : (Q) \rightarrow \Gamma$ is the transition function
6. **Initial State**: $q_0$ is an initial state

---

**Moore Machine**

A Mealy machine is a finite state automata transducer for which the output is based on state. The output symbol is based on the state the system transitions to. Each state has an associated output symbol which is asserted by the system when that state is transitioned into. The initial state does not produce an output symbol in the first step, but produces its output symbol anytime after that the system enters that state.

**Example 3.12.** Consider the automata shown in Figure 3.6.

![Moore Machine Diagram](image)

**Figure 3.6: Moore Machine**

What we notice about this diagram is that each state has a label with two symbols of the type $a/b$. The symbol $a$ denotes the state, and symbol $b$ belongs to the output alphabet. The Figure shows that when the machine enters state $q_0$ (not in the initial step) an output of $e$ is obtained, and similarly, when the machine enters state $q_1$ an output of $o$ is obtained. In other words the output function is $y(q_0) = e$ and $y(q_1) = o$. There is no final state. The machine operates till the last symbol of the input string is encountered. Let us take an arbitrary input string and ascertain its output from this machine. If we take 01101 as the input, we start in state $q_0$, for the first 0, we stay in $q_0$ and therefore get $y(q_0) = 0$. Next we get 1 as the input and we transition to state $q_1$, and therefore, the output for that is $y(q_1) = o$. Following similarly, the output string we get is $eoeeo$. The machine is a parity checker for the number of 1s encountered from the beginning to the current time. The state $q_0$ shows even
parity, and the state $q_1$ shows odd. We can show this machine transforming input string to output string as:

\[ 01101 \rightarrow M \rightarrow eoeeo \]

Formally, the Moore machine is given by:

**Definition 3.13. Moore Machine**: A Moore machine is a sextuple $(Q, \Sigma, \Gamma, \delta, y, q_0)$, where

1. **States**: $Q$ is a finite set of states
2. **Input Alphabet**: $\Sigma$ is a finite set of input symbols, called the input alphabet
3. **Output Alphabet**: $\Gamma$ is a finite set of output symbols, called the output alphabet
4. **Transition Function**: $\delta : (Q \times \Sigma) \rightarrow Q$ is the state transition function
5. **Output Function**: $y : (Q \times \Sigma) \rightarrow \Gamma$ is the transition function
6. **Initial State**: $q_0$ is an initial state

---

**Equivalence and Limitations of Finite State Transducers**

Mealy and Moore machines are equivalent. This means that for any given Mealy or Moore machine, one can design a corresponding Moore or Mealy machine respectively, such that for every input string to both machines, the output strings will be identical.

If we fix all the input strings applied to a Finite State Transducer (FST), such as a Mealy or Moore machine, to be belonging to a regular language, i.e. the strings are accepted by a finite state machine, or equivalently, they are produced by another FST, then the strings that the FST will produce will also belong to some regular language. This can be proven by combining the generating and the transducer automata into a single automaton (see [DDQ78] for details).

The computable functions one can build with these machines are limited due to their finiteness of states. For instance Minsky [Min67] showed that they are not capable of multiplication. Finite state automata and machines, however are very useful in designing systems with finite memory such as digital circuits with memory, as well as lexical analyzers for language compilers that evaluate tokens such as constants, variable names, etc.
3.2.3 Regular Languages

Regular expressions are a way to describe regular languages. Let us look at a regular expression and then find out its corresponding language. Consider \((a + b)^* ab\). Think of the symbol + as a symbol union of sets, * as the Kleene star closure, obtaining every string including the null string, \(\lambda\), and symbols next to each other as concatenation. Each single symbol can be viewed as a singleton set. For instance \(a\) should be viewed as \(\{a\}\). Let’s study this example stepwise as shown below.

**Example 3.14.** Consider the regular expression \((a + b)^* ab\) that we will convert into a language.

\[
(a + b)^* ab \Rightarrow \{a \cup b\}^* \{\{a\}\{b\}\} \\
\Rightarrow \{a, b\}^* ab \\
\Rightarrow \{\lambda, a, b, aa, ab, ba, bb, \cdots\}ab \\
\Rightarrow \{ab, aab, bab, aaab, abab, baab, bbab, \cdots\}
\]

Regular expressions completely characterize regular languages, i.e. for every regular expression, its corresponding language is accepted by a deterministic finite automata making it a regular language, and conversely, every regular language has a corresponding regular expression. Please refer to [Lin11] or [Sud05] for the proofs of both these statements. This relationship between regular expressions (related to regular sets) and deterministic finite automata are given precisely by Kleene’s theorem (see [Sud05]).

**Definition 3.15** (Regular Sets). Regular sets are sets that include some basic elements and then all elements that can be produced recursively by performing union, concatenation, and Kleene star operations.

**Theorem 3.16** (Kleene’s Theorem). A language is accepted by a deterministic finite automata if and only if it is a regular set over the same alphabet.

3.2.4 Regular Grammars

Regular languages are generated by right-linear and left-linear grammars. Definition of a linear grammar is given below.

**Definition 3.17.** A grammar whose each production rule has at most one non-terminal on its right side is called a linear grammar.
A right linear grammar has a variable replaced by either a string of terminal symbols, or a string of terminal symbols followed on the right by a non-terminal symbol.

**Definition 3.18.** A grammar \((N, T, S, \mathcal{P})\) whose all production rules are of the type shown in Table 3.2 is called a **right linear grammar**.

<table>
<thead>
<tr>
<th>Substitution</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\langle C \rangle ::= c )</td>
</tr>
<tr>
<td>(\langle A \rangle ::= a \langle B \rangle )</td>
</tr>
</tbody>
</table>

Here \(A, B, C \in N\) and \(a, c \in T^*\).

Similarly, a left linear grammar has a variable replaced by either a string of terminal symbols, or a nonterminal on the left followed by a string of terminal symbols.

**Definition 3.19.** A grammar \((N, T, S, \mathcal{P})\) whose all production rules are of the type shown in Table 3.3 is called a **left linear grammar**.

<table>
<thead>
<tr>
<th>Substitution</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\langle C \rangle ::= c )</td>
</tr>
<tr>
<td>(\langle A \rangle ::= \langle B \rangle a )</td>
</tr>
</tbody>
</table>

Here \(A, B, C \in N\) and \(a, c \in T^*\).

### 3.2.5 Closure Properties of Regular Languages

**Theorem 3.20.** If \(L_1\) and \(L_2\) are regular languages, then \(L_1^*\), \(L_1^c\), \(L_1 L_2\), \(L_1 \cup L_2\), and \(L_1 \cap L_2\) are all regular languages.
3.2. FINITE STATE AUTOMATA, REGULAR LANGUAGES, AND GRAMMARS

Proof. The proof for the set operations of union, and also for concatenation and star closure come from the corresponding properties of the regular expressions expressing those languages. For complement language, just change the accepting set of states to be the complement of the ones for a language, and we obtain an automata accepting the complement making it a regular language. The intersection can be proven by constructing a new automata from the intersecting languages.

3.2.6 Decidable Properties of Regular Languages

Theorem 3.21. Given a grammar for a regular language \( L \) on an alphabet \( \mathbb{A} \) and a string \( \omega \in \mathbb{A}^* \), then an algorithm in a finite number of steps can ascertain if \( \omega \in L \) is true or false.

Proof. Construct a DFA for the language, and then execute the DFA with the input string \( \omega \). In finite steps (the length of the string) it will be known if the string belongs to the language or not depending on if the DFA terminates in the set of final states or not.

Theorem 3.22. It is decidable if a given regular grammar has a language that is empty, finite, or infinite.

Proof. The graph of the DFA allows us to ascertain if there’s a path from start to the set of final states or not, and if any path contains any loops.

Theorem 3.23. It is decidable if two given regular languages are equal.

Proof. Given \( L_1 \) and \( L_2 \), we see that \((L_1 \cap L_2^c) \cup (L_1^c \cap L_2) = \phi \iff L_1 = L_2 \).

3.2.7 A Nonregular Language

Any language that requires a non-finite memory can not be regular, since no finite state automata will be able to recognize it. hence, if we fix an automaton of size (number of nodes \( n \)), then if the memory required is more than this fixed \( n \), then the language can not be recognized and hence will be non-regular. The proof of this depends on using a Pumping lemma, which uses the pigeonhole principle, essentially stating that for an automata with \( n \) states any string more than length \( n \) has to repeat a node.

The language \( L = \{a^n b^n|n \geq 0\} \) is non-regular. The reason this language is non-regular is that as a machine processes the coming string it needs to remember how many \( a \)'s it has encountered. However, the number of \( a \)'s the
string can have is not bounded. Hence, if the machine has a fixed number of states, it’s memory can not be larger than that, and it will not be able to remember number of a’s larger than that size.

Another way to prove this statement is by using the pumping lemma for regular languages which is presented next.

**Pumping Lemma for Regular Languages**

To understand how the pumping lemma works and how it can be used to prove nonregularity of languages, we use the deterministic finite automaton in Figure 3.7.

![Figure 3.7: Pumping Lemma for Regular Languages](image)

We see that there are four states in the automaton shown in Figure 3.7. Hence, any string that has more than 4 symbols has to visit some state more than once. That means that the traversing of the transition graph will have at least one loop from that state to itself. For instance, let’s study the string \(abbaba\). This string has length 6 which is more than 4 and hence some state must be repeated. Let’s look at its transitions, shown in Equation 3.9.

\[
q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_2 \xrightarrow{a} q_3
\]  

(3.9)

We see the loop \(bba\) in this string given by

\[
q_1 \xrightarrow{b} q_2 \xrightarrow{b} q_0 \xrightarrow{a} q_1
\]  

(3.10)

We can see the transitions as shown here.
3.3. PUSHDOWN AUTOMATA, CONTEXT-FREE LANGUAGES, AND GRAMMARS

It is clear that the string \(a(bba)^nba\) for \(n \geq 0\) must also belong to the same language. This is what the pumping lemma says for a regular language. It states that there must be a substring that can be pumped as many finite number of times and the resulting string must still be in the language. Hence a string of the type \(wxy\) in the language where the string \(x\) is the one obtained from a loop such as the \(bba\) string in our example, then \(wx^n y\) must also be in the language.

We are ready now to formally state the pumping lemma. For any string \(s\), \(|s|\) will indicate its length, i.e. how many non-null symbols it contains. For instance \(|abba| = 4\).

**Theorem 3.24.** For each regular language \(\mathcal{L}\), there exists a fixed pumping length \(\ell\) such that for any \(s \in \mathcal{L}\), for which \(|s| \geq \ell\), we can divide the string into three pieces as \(s = xyz\), such that

1. \(|y| > 0\)
2. \(|xy| \leq \ell\). and
3. \(\forall i \geq 0, xy^i z \in \mathcal{L}\).

How this pumping lemma is used to prove that the language \(a^n b^n\) is not regular is as follows. If it were regular, then we would be able to pump some substring. Let us assume that the entire pumping string is made up of some finite number of \(a\). However, then we would be able to pump this substring, which would make the string have more \(a\) symbols then how many we would have of \(b\), and then we wouldn’t get \(a^n b^n\). Similar argument is also valid if the entire pumping string consisted of only \(b\) symbols. The only possibility is that the pumping string would have some \(a\) and some \(b\), but pumping that would again not give \(a^n b^n\). Hence \(a^n b^n\) can not be regular.

3.3 Pushdown Automata, Context-Free Languages, and Grammars

We saw that some languages are not accepted by finite state automata. The languages that can not be accepted by finite state automata are the ones that require infinite memory. Let’s take the language \(\mathcal{L} = \{a^n b^n | n \geq 0\}\), and add an infinite memory to our machine so that this language can be recognized. We will accomplish this by adding an infinite deep stack to our automata. Then
we will find the grammar that is needed to generate the language that this modified machine recognizes.

### 3.3.1 Pushdown Automata

Let’s take the language $\mathcal{L} = \{a^n b^n | n \geq 0\}$ and modify our finite state machine by adding infinite storage to it. We will study Figure 3.8 and Figure 3.9 which show the dynamics of the Pushdown Automaton (PDA) and its interaction with the input tape and the infinite storage which is a pushdown stack.

![Figure 3.8: PDA: $a^n b^n$ Acceptor](image)

As we see in Figure 3.9, the PDA, just like a finite state automaton, also has an input tape with the finite input string on it. Similarly, it also has a read-head to read the input symbols one at a time by moving to the right each step. What it has more than the DFA is a stack. Stack is an infinite long storage mechanism which works on a First-In-Last-Out (FIFO) basis. The PDA can push stack symbols on it from the top, pushing all of them downwards, and it can also pop the top most symbol out which makes the stack symbols shift up.

How the transitions for this PDA work are shown in its transition graph in Figure 3.8. The nodes of the graph indicate the current state as before. The arcs however show more information. The arc label is a triplet as the transition from state $q_0$ to $q_1$ shows. We see the label as $a, 0, 1$. This directed arc from $q_0$ to $q_1$ indicates that when the system is in state $q_0$ and the head reads the input symbol $a$ and the stack symbol on the top of the stack is 0, then this input can enable the following two actions. The head will push the stack symbol 1 on top of the stack (pushing the rest of the stack down one step) and the state will change to $q_1$. Hence the transition function takes three inputs, the current state, such as $q_0$, the input symbol, $a$ in this example, and the top of the stack symbol, which is 0 here and produces two actions; one either a
push of a symbol onto the top of the stack, or a pop off of a symbol from the top, and the second being the state transition. This transition is represented as $\delta(q_0, a, 0) = (q_1, 1)$. As an example of popping a symbol off from the stack, let us look at the transition from state $q_1$ to $q_2$, where the arc label is $b, 1, \lambda$. The symbol $\lambda$ as the third symbol of the label indicates that the top of the stack symbol will be popped off from the stack. This transition is represented as $\delta(q_1, b, 1) = (q_2, \lambda)$. Notice that the PDA is a non-deterministic PDA. The reason for that is that there is no transition from state $q_0$ for the input symbol $b$ (so it has incomplete transition information), it has a $\lambda$-transition from $q_0$ to $q_3$ with the arc label $(\lambda, 0, \lambda)$, the first symbol showing that without any in-
put symbol, there can be a transition, and finally on the same arc we see two transitions.

We present the definition of the nondeterministic pushdown automata formally as

**Definition 3.25. Nondeterministic Pushdown Automata (NPDA):** A nondeterministic pushdown automata is a septuple \((Q, \Sigma, \Theta, \delta, q_0, s, F)\), where

1. **States:** \(Q\) is a finite set of states
2. **Input Alphabet:** \(\Sigma\) is a finite set of input symbols, called the input alphabet
3. **Stack Alphabet:** \(\Theta\) is a finite set of stack symbols, called the stack alphabet
4. **Transition Function:** \(\delta: ([Q \cup \lambda] \times \Sigma \times \Theta) \rightarrow 2^{(Q \times \{\Theta \cup \{\lambda\}\)}}\) is the transition function
5. **Initial State:** \(q_0\) is a initial state
6. **Initial Stack Symbol:** \(s \in \Theta\) is the symbol on the top of the stack initially
7. **Final Set of States:** \(F\) is the set of final or terminal states

\[\square\]

**Example 3.26.** For our example we go back to the NPDA shown in Figure 3.8 and Figure 3.9. The details of the septuple for this example are shown in Table 3.4.

<table>
<thead>
<tr>
<th>NPDA Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Q)</td>
</tr>
<tr>
<td>(\Sigma)</td>
</tr>
<tr>
<td>(\Theta)</td>
</tr>
<tr>
<td>(q_0)</td>
</tr>
<tr>
<td>(s)</td>
</tr>
<tr>
<td>(F)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\delta(q_0, a, 0))</td>
</tr>
<tr>
<td>(\delta(q_0, \lambda, 0))</td>
</tr>
<tr>
<td>(\delta(q_1, a, 1))</td>
</tr>
<tr>
<td>(\delta(q_1, b, 1))</td>
</tr>
<tr>
<td>(\delta(q_2, b, 1))</td>
</tr>
<tr>
<td>(\delta(q_2, \lambda, 0))</td>
</tr>
</tbody>
</table>

Table 3.4: NPDA Accepting \(a^n b^n\)
3.3. PUSHDOWN AUTOMATA, CONTEXT-FREE LANGUAGES, AND GRAMMARS

Since the automaton is a nondeterministic one, we just need to find a sequence of allowed transitions from the start to the final step where we end up in a final state for any string of type \( a^n b^n \). Figure 3.9 shows these steps clearly, and corresponding to each step, the transition of that can be checked in Figure 3.8, as well as in the Table 3.4. We can see that the system starts in state \( q_0 \) with stack symbol 0 on top, and when it reads the input symbol \( a \) on the tape, it applies the transition \( \delta(q_0, a, 0) \) with the output \( (q_1, 1) \), which means that its state changes to \( q_1 \) and it pushes 1 on top of the stack. Similarly, the other steps take place as shown, and after reading \( aabb \), it makes the \( \lambda \) transition to the final state and accepts the string.

3.3.2 Context-Free Grammars and Languages

Just as regular expressions, grammars and languages were completely characterized by finite state automata, languages that are called context-free, which are obtained by context-free grammars are completely characterized by non-deterministic pushdown automata. Now, we will define what context-free grammar is.

**Definition 3.27.** A grammar \((N, T, S, P)\) whose all production rules are of the type shown in Table 3.5 is called a **context-free grammar**.

<table>
<thead>
<tr>
<th>Substitution</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \langle A \rangle ::= v )</td>
</tr>
</tbody>
</table>

Here \( A \in N \) and \( v \in (N \cup T)^* \).

**Example 3.28.** Consider the grammar shown in Table 3.6.

<table>
<thead>
<tr>
<th>Substitution</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \langle S \rangle ::= a \langle S \rangle b</td>
</tr>
</tbody>
</table>
This grammar generates the language $L = \{a^n b^n | n \geq 0\}$. We have also kept a $\lambda$ production rule in the grammar to produce the null string. △

The language $L = \{a^n b^n | n \geq 0\}$ is important in programming languages, because it represents the form of matching left parentheses with right ones in expressions, such as $(x + (y + (2 \times 5 - x)))$. The left parentheses are represented by $a$ and the right ones by $b$ in $a^n b^n$, and the number of $a$ symbols have to be the same as the number of $b$ symbols. The actual language needed in fact is slightly more complicated, since we can have $(x + y)(3 + (5 + 6))$. This does not have $a^n b^n$ form. In this case, the two symbols still have to match and at an prefix of the final string or expression the number of $b$ symbols have to equal or less than the number of $a$s for that prefix.

In evaluating expressions, not only the syntax is important, i.e. if the expression belongs to the language, but also the semantics of the expression (the meaning). We will make this clear by examining an expression and using derivation trees to evaluate the expression.

Consider the expression $x + y \times z$. This expression can have two interpretations or meanings (semantics), one where we perform the $+$ operation first and the other where we perform the $\times$ operation first. For instance $5 + 2 \times 3$ will be 21 in the first case and 11 in the second case, hence having completely different meaning.

Consider the two different derivation trees for this expression shown in Figure 3.10. The letters in the tree nodes stand for the following: $E$ for Expression, $I$ for Item, $+$ for addition, $\times$ for multiplication, and finally, $x$, $y$, and $z$ for the terminal symbols.

![Figure 3.10: Two Derivation Trees for $x + y \times z$](image)

The root (on top) of both trees is the symbol $E$. In the tree on the left, the root $E$ is replaced by $E + E$. The left $E$ gets replaced by $x$, while the right $E$
after two more levels gets replaced by $x \times y$. The expression then semantically performs the operation $x + (y \times z)$. These derivations can be shown as

$$E \Rightarrow E + E \Rightarrow x + E \Rightarrow x + E \times E \Rightarrow x + y \times E \Rightarrow x + y \times z \quad (3.11)$$

Notice that the sequential form of the derivation shown in Equation 3.11 expands the left most non-terminal in every step. We could have also done a rightmost sequential derivation for the same semantically meaning left derivation tree of Figure 3.10. These both sequential derivation would however evaluate to the same answer. So, the semantics of the expression are tied to its derivation tree and not its sequential derivation directly. The sequential form of the derivation shown in Equation 3.12 expands the right most non-terminal in every step.

$$E \Rightarrow E + E \Rightarrow E + E \times E \Rightarrow E + E \times z \Rightarrow E + y \times z \Rightarrow x + y \times z \quad (3.12)$$

The right derivation tree in Figure 3.10 is semantically different from the left one, although, they both can be derived from the same grammar shown in Table 3.7.

### Table 3.7: Grammar for Arithmetic Expressions

<table>
<thead>
<tr>
<th>Sets</th>
<th>Substitutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N = {E}$</td>
<td>$\langle E \rangle ::= \langle E \rangle + \langle E \rangle</td>
</tr>
<tr>
<td>$T = {x, y, z, +, \times, (, )}$</td>
<td></td>
</tr>
</tbody>
</table>

To fix the ambiguity in the semantics of this grammar, we can enforce precedence rules for the arithmetic operation, such as multiplication having higher precedence than addition. This can be used to design a new grammar which will be non-ambiguous for this example. This non-ambiguous grammar is shown in Table 3.8. This grammar, the way it has been designed has in fact both + and * left-associative, and * has precedence over +. Left associativity of both operator means that an expression like $x + y \times x + z$ is evaluated as $((x + y) \times x) + z$. 
Table 3.8: Non-ambiguous Grammar for Arithmetic Expressions

<table>
<thead>
<tr>
<th>Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$ = ${E, T, F}$</td>
</tr>
<tr>
<td>$T$ = ${x, y, z, +, \times, (, )}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Substitutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\langle E \rangle ::= \langle E \rangle + \langle T \rangle</td>
</tr>
<tr>
<td>$\langle T \rangle ::= \langle T \rangle \times \langle F \rangle</td>
</tr>
<tr>
<td>$\langle F \rangle ::= \langle (E) \rangle</td>
</tr>
</tbody>
</table>

The only derivation tree for the unambiguous grammar in Table 3.8 is shown in Figure 3.11.

**Definition 3.29.** A context free language for which no unambiguous grammar exists is called inherently ambiguous, otherwise, the language is called unambiguous.

An example language that is inherently ambiguous is the one shown in Equation 3.13 by a proof in [Har78], by analyzing the case for $n = m$.

\[
\mathcal{L} = \{a^n b^r c^m\} \cup \{a^n b^m c^m\} \text{ for } n, m \geq 0 \quad (3.13)
\]
3.3. PUSHDOWN AUTOMATA, CONTEXT-FREE LANGUAGES, AND GRAMMARS

Context free languages can be put in many different normal forms. Two of the well studied ones are the Chomsky normal form and the Greibach normal form. For these two forms, for any context free language, for which \( \lambda \) doesn't belong to the language, an equivalent grammar exists in these forms that generates the same language. These two forms are presented below.

**Definition 3.30.** If all the production rules of a context free grammar \( (N, T, S, \mathcal{P}) \) are of the type \( \langle E \rangle ::= \langle F \rangle \langle G \rangle \), or \( \langle H \rangle ::= x \), where \( \langle E \rangle, \langle F \rangle, \langle G \rangle, \langle H \rangle \in N \), and \( x \in T \), then the grammar is in the **Chomsky Normal Form** (CNF).

**Definition 3.31.** If all the production rules of a context free grammar \( (N, T, S, \mathcal{P}) \) are of the type \( \langle E \rangle ::= x \langle Y \rangle \), where \( \langle E \rangle \in N \), \( x \in T \), and \( \langle Y \rangle \in N^* \), then the grammar is in the **Greibach Normal Form** (GNF).

**Membership and Parsing Complexity for Context Free Languages**

There is an algorithm (CYK algorithm) that parses and solves the membership problem for a string \( s \) in a context free language with complexity \( O(n) \), \( n \) being the size of the string, when we have the grammar in the CNF.

Although general context free languages can be parsed and their membership solved in \( O(n^3) \) time, but there are restricted types of these languages for which parsing is much more efficient. For instance a restricted context free grammar can be parsed in linear time if it is a simple or s-grammar defined below.

**Definition 3.32.** If all the production rules of a context free grammar \( (N, T, S, \mathcal{P}) \) are of the type \( \langle E \rangle ::= x \langle Y \rangle \), where \( \langle E \rangle \in N \), \( x \in T \), and \( \langle Y \rangle \in N^* \), and moreover, the pair \( (\langle E \rangle, x) \) does not occur more than once in the productions \( \mathcal{P} \), then the grammar is an **s-grammar**.

Another class of restricted context free language is obtained from what are called Deterministic Pushdown Automata (DPDA). These are defined as follows.

**Definition 3.33.** **Deterministic Pushdown Automata (DPDA):** A deterministic pushdown automata is a NPDA with the restrictions that:

1. there is at most one transition for a given input symbol and the top stack symbol, and
2. if a \( \lambda \) move is possible in some state for a given stack top symbol, there should be no transition available for the same state and the stack top symbol with any other input available.
There are efficient parsers available for deterministic context free languages as there is no need for backtracking, as only one move is allowed for a given state and input. The grammars associated with these include s-gra mmar, and also LL grammar, LL(k) grammars, and LR grammars that prove very beneficial in programming languages.

3.3.3 A Non Context Free Language

Any language that requires a non-finite memory can not be regular, since no finite state automata will be able to recognize it. hence, if we fix an automaton of size (number of nodes $n$), then if the memory required is more than this fixed $n$, then the language can not be recognized and hence will be non-regular. The proof of this depends on using a Pumping lemma, which uses the pigeonhole principle, essentially stating that for an automata with $n$ states any string more than length $n$ has to repeat a node.

The language $L = \{a^n b^n | n \geq 0\}$ is non-regular. The reason this language is non-regular is that as a machine processes the coming string it needs to remember how many $a$’s it has encountered. However, the number of $a$’s the string can have is not bounded. Hence, if the machine has a fixed number of states, it’s memory can not be larger than that, and it will not be able to remember number of $a$’s larger than that size.

Another way to prove this statement is by using the pumping lemma for regular languages which is presented next.

Pumping Lemma for Context Free Languages

To understand how the pumping lemma for context free languages works and how it can be used to prove that a language is not context free, we use the derivation tree in Figure 3.12. Regular languages were characterized by finite number of states, and that is why they couldn’t handle $a^n b^n$. Context free languages are more powerful, and they have infinite state because of the infinite stack. However, because of their restriction on the structure of their grammar, if we remove all $\lambda$-production and get an equivalent grammar (the symbol $\lambda$ can trivially be added to the language if we needed a language with $\lambda$, then it shows that we have non-decreasing productions as far as lengths of strings (and partial strings, during the execution of the grammar rules) are concerned. Hence, this should give us certain behavior. That specific behavior shows up in the pumping lemma for context free languages as follows.

Although context free languages don’t have finite states, but they do have a finite number of symbols. Hence, if it is an infinite language, and the length of the string is more than how many variables there are, then some variable
will be repeated. Let us study Figure 3.12 carefully keeping this in mind. The figure shows a partial derivation tree. It shows that we can replace $S$ by $vEz$. We also see that $E$ can be replaced by $aB$, which gives us $acDd$, and which finally gives us $S$ replaced by $vacEf$. If we make the $E$ go through again with its derivation as shown by the dashed line, we will get $v(ac)^2E(hf)^2z$. One more round will give $v(ac)^3E(hf)^3z$. We can get as many repeats as we want. Let’s say then from here $E$ can be replaced by the string $qr$, then our total derivation from $S$ would give us $S = vacqrhfz$. Call this string $s$, and we see that $s = vacqrhfz$, and we use, $w = ac$, $x = qr$, $y = hf$, then, we could write $s = vwxy$, and moreover, as we have seen, we would have, for every $i \geq 0$, $vw^i xy^i z$ would be derivable through this derivation tree, and hence would belong to the same language. We present this formally as the pumping lemma for context free languages.

Figure 3.12: Pumping Lemma for Context Free Languages

**Theorem 3.34.** For each context free language $\Sigma$, there exists a fixed pumping length $\ell$ such that for any $s \in \Sigma$, for which $|s| \geq \ell$, we can divide the string into five pieces as $s = vwxyz$, such that

1. $|wy| > 0$
2. $|wxy| \leq \ell$, and
3. $\forall i \geq 0, vw^i xy^i z \in \Sigma$.

The application of the pumping lemma for context free languages shows that the language $a^n b^n c^n$ is not a context free language. The reason is that the pumping lemma can only pump two pieces synchronously, but this language
has three pieces that need to do the same. So, just like we used the pumping lemma for proving the existence of a non-regular language, this pumping lemma can be applied to this case here, by taking different parts as the pumping part and realizing then, that this language can not context free.

3.3.4 Closure Properties of Context Free Languages

Theorem 3.35. If $L_1$ and $L_2$ are context free languages, then $L_1^*$, $L_1L_2$, and $L_1 \cup L_2$ are all context free languages.

Proof. Given two context free languages, we can create a new one which is the union of the two. This is easily accomplished by creating a new starting symbol for the new language, adding two production rules, one each for taking the new starting symbol to the starting symbols of the other two, and finally using all production rules for both languages in the new language including the union of all final states. This new language is the union of the two. If $\langle S_1 \rangle$ and $\langle S_2 \rangle$ are the starting symbol for $L_1$ and $L_2$, and $\langle S_3 \rangle$ for $L_3$, then the new rules added are $\langle S_3 \rangle := \langle S_1 \rangle \mid \langle S_2 \rangle$. Here, we get $L_3 = L_1 \cup L_2$.

For concatenation, if $\langle S_1 \rangle$ and $\langle S_2 \rangle$ are the starting symbol for $L_1$ and $L_2$, and $\langle S_3 \rangle$ for $L_3$, then the new rules added are $\langle S_3 \rangle := \langle S_1 \rangle \langle S_2 \rangle$. Here, we get $L_3 = L_1L_2$.

For star closure of $L_1$, we need to add the language $L_1^0$, since we already know that all concatenations of $L_1$ are context free. If $\langle S_1 \rangle$ is the starting symbol for $L_1$ and $\langle S_2 \rangle$ for $L_2$, then the new rules added are $\langle S_2 \rangle := \langle S_1 \rangle \langle S_2 \rangle \mid \lambda$. Here, we get $L_2 = L_1^*$.

Lemma 3.36. The language $\mathcal{L} = \{a^n b^n c^m \mid n \geq 0, m \geq 0\}$ is a context free language.

Proof. We see that $\mathcal{L} = L_1L_2$, where $L_1 = \{a^n b^n \mid n \geq 0\}$ and $L_2 = \{a^n \mid n \geq 0\}$, and since both languages are context free, and context free languages are closed under concatenation, we have the result.

Theorem 3.37. If $L_1$ and $L_2$ are context free languages, then $L_1 \cap L_2$ and $L_1^c$ are in general not guaranteed to be context free languages, i.e. context free languages are not closed under intersection and complementation.

Proof. The language $\mathcal{L} = \{a^n b^n c^n \mid n \geq 0\}$ is not a context free language. The languages $L_1 = \{a^n b^n c^m \mid n \geq 0, m \geq 0\}$ and $L_2 = \{a^n b^m c^m \mid n \geq 0, m \geq 0\}$ are context free languages. Since $\mathcal{L} = L_1 \cap L_2$, the result is proven.
3.3. PUSHDOWN AUTOMATA, CONTEXT-FREE LANGUAGES, AND GRAMMARS

Theorem 3.38. Context free languages are closed under regular intersection, i.e. the intersection of a context free language and a regular language is again context free.

Proof. Given the two languages, one has a deterministic finite automata attached with it, and the other has an NPDA. We create a product state space of the states of both, and create a new NPDA, which simulates both machines and accepts a string if that string is accepted by both.

3.3.5 Decidable Properties of Context Free Languages

Theorem 3.39. Given a context free grammar, it is decidable if its corresponding language is empty.

Proof. Starting with all the terminals, without loss of generality, assuming a $\lambda$-free language, we iteratively include all non-terminals that can be transformed into terminals and the non-terminals that can lead to terminals using some grammar production rules. If the starting symbol does not belong to this set, the language is empty.

Theorem 3.40. Given a context free grammar, it is decidable if its corresponding language is infinite.

Proof. We can remove $\lambda$ productions in the grammar, remove all useless production rules, and also unit-productions (i.e., one variable transforming to another) and still obtain an equivalent grammar. A variable is useless if it can never lead to a string. Any production involving useless variables is useless. A $\lambda$-production is a rule of type $\langle A \rangle ::= \lambda$. For a $\lambda$-free language we can obtain an equivalent grammar not having any $\lambda$-productions. The algorithm for this transformation starts with $\lambda$ and moves backwards to find all non-terminals leading to $\lambda$ iteratively in a way very similar to the way useless variables and productions are found. Unit productions are rules of the kind $\langle A \rangle ::= \langle B \rangle$. We can remove the right side by the right hand side of rules where $\langle B \rangle$ is on the left, except for those where that will lead back to $\langle A \rangle$. The details of these three transformations can be obtained from many standard sources, such as [Lin11]. Draw a dependency graph for the variables in this grammar so that there is an arc from a variable $A$ to $B$, if $A$ is on the left of a production rule, and $B$ is on the right. Existence of a cycle indicates infinite language and vice versa.
Theorem 3.41. Given a context free grammar and a string, it is decidable whether the string belongs to the language or not.

Proof. We can write a context free grammar in the Chomsky normal form. Using that form any string of length \( n \) is derivable in \( 2^n - 1 \) steps. Hence, given a string of length \( n \), in finite number of steps, it is determined if the string belongs to the language or not.

Another way to see this is by noting that a pushdown automaton will use one step per input symbol. Hence, in finite number of steps it will have reached the end of the input, where it can be decided if a final state has been achieved or not, and hence, if the string is accepted or not.

3.4 Turing Machines and Languages

We have already seen an example of Turing machine in section 2.4.5 of Chapter 2. We start here with formally defining what a Turing machine is and then we study languages and grammar related to Turing machines.

3.4.1 Turing Machines

Definition 3.42. Turing Machine (TM): A Turing machine is a septuple \((Q, \Sigma, \Theta, \delta, q_0, \square, F)\), where

1. **States:** \( Q \) is a finite set of states
2. **Input Alphabet:** \( \Sigma \) is a finite set of input symbols, called the input alphabet
3. **Tape Alphabet:** \( \Theta \) is a finite set of tape symbols, called the tape alphabet
4. **Transition Function:** \( \delta : \Omega \subset Q \times \Theta \rightarrow Q \times \Theta \times \{L \times R\} \) is the transition function
5. **Initial State:** \( q_0 \) is a initial state
6. **Blank Tape Symbol:** \( \square \) is the blank tape symbol which is used to show the beginning and end of the input
7. **Final Set of States:** \( F \) is the set of final or terminal states

We will compare the differences between finite state automata, pushdown automata, and Turing machines. Please refer to Figure 3.13 for this comparison. Subfigure 3.13a shows the finite automata, subfigure 3.13b shows the pushdown automata, and subfigure 3.13c shows the Turing machine.

The finite automata and the pushdown automata, both have read only heads. Both of these make one shift to the right after every transition, and hence finish processing the input in \( n \) steps, where \( n \) is the length of the input.
3.4. TURING MACHINES AND LANGUAGES

String. Turing machine, on the other hand, reads an input from the current cell of the tape, then writes on the current cell of the tape, changes its state, and moves either left or write. It’s tape is of infinite length. These three different types of transitions can be seen in Figure 3.14.

The transition graph of the finite automata in Figure 3.14a shows $\delta(q_0, a) = q_1$, which shows the transition of the automata from state $q_0$ to $q_1$ when the input is $a$. The transition graph of the pushdown automata in Figure 3.14b shows $\delta(q_0, a, 0) = (q_1, 10)$, which shows the transition of the pushdown au-

Figure 3.13: Comparing DFA, NPDA, and TM

Figure 3.14: Comparing Transition Graphs for DFA, NPDA, and TM
tomata from state \( q_0 \) to \( q_1 \) when the input is \( a \), the stack symbol on top is 0, and the pushdown automata pushes 1 on top of the stack. The transition graph of the Turing machine in Figure 3.14c shows \( \delta(q_0, a) = (q_1, b, R) \), which shows the transition of the Turing machine from state \( q_0 \) to \( q_1 \) when the input on the current tape cell is \( a \), and the Turing machine writes the symbol \( b \) on the current cell of the tape and slides one cell to the right. It seems that Turing machine can write, and hence it transforms input strings to output strings and hence, it is a transducer. However, we have already seen that the other machines can also be transducers. So, the most important difference, which creates a dramatic difference in capabilities is that the read/write head of the Turing machine can move both ways. This is significant, because for the other two, they could only perform finite number of operations for a fixed length input string since they moved one step to the right for each symbol. Turing machine on the other hand can do, in fact unbounded number of steps for each input symbol. This leads to the question of \textit{halting} in a Turing machine and how that is the essential element of its operation.

Some very important points to pay attention to in Turing machines are the following:

- The Turing machine tape is infinite
- The read-write head reads the symbols and also writes on the tape
- The read-write head can move left and right
- The transition function does not map the entire \( Q \times \Theta \) product space, but only a subset of that. This means that there are configurations in \( Q \times \Theta \) where the Turing machine halts.
- For a \textbf{standard deterministic Turing machine} there is a single output for every member of the domain \( \Omega \) of the transition function.
- We make the assumption in our design that when the Turing machine reaches any final state, it halts, i.e. there are no transitions from any of the final states.

Let us understand the operation of a Turing machine via some examples.

\textbf{Example 3.43.} Given a transition \( \delta(q_0, a) = (q_1, b, L) \) for a Turing machine, we will go through the transition as shown in Figure 3.15. The machine is in state \( q_0 \), it reads the input \( a \), changes it to \( b \), changes state to \( q_1 \) and then shifts left.

\( \triangle \)

Let us say that a Turing machine transitions from the configuration shown in the left to that shown on the right in Figure 3.16.

We show the transition for a Turing machine \( \mathcal{T} \) as

\[ a_0 q_0 a_1 a_2 a_3 a_4 \Rightarrow_{\mathcal{T}} a_0 a_1 q_1 a_2 a_3 a_4 \]  

(3.14)
3.4. TURING MACHINES AND LANGUAGES

If, on the other hand the Turing machine state $a_0q_0a_1a_2a_3a_4$ transitions to the state $a_0a_1q_1a_2a_3a_4$ in multiple steps, we show that as

We show the transition for a Turing machine $T$ as

$$a_0q_0a_1a_2a_3a_4 \Rightarrow_T a_0a_1a_2a_3a_4$$

(3.15)

The state $a_0q_0a_1a_2a_3a_4$ means that the Turing machine is in state $q_0$ and the read-write head is on the symbol $a_1$ on the tape, where the other symbols on the tape are also shown.

Let us now understand the operation of a Turing machine.

Turing Machine Operation Steps

We will enumerate the steps of the operation of a Turing machine.

1. **Input**: The initial configuration of a Turing machine is
   $\square q_0a_0a_1a_2a_3a_4 \cdots a_n \square$, where the input string is $a_0a_1 \cdots a_n$, the initial state is $q_0$ and the read-write head is placed to the left of the symbol $a_0$ of the input.

2. **Halting or no-halting**: The Turing machine starting from the initial configuration starts going through transitions, and then either it will halt, or it will keep transitioning for ever and never halt.

3. **State at Halting**: If the machine halts, then the state of the machine at that configuration decides if the input has been accepted or not depending on if that state is a final state or not.

4. **Output String**: If the machine halts, then the string on the tape is the output string of the Turing machine transducer.
What makes Turing machine different from the other machines is its ability to move in both directions. Even if we make a Turing machine with the tape being infinite only on one side and terminating at the other end, it would still have the same computational power. Hence, its power, which, in fact makes it a universal computation machine (a computer) is because of its dual movement. However, that same power allows it to reach halting condition which wasn’t present in the other machines. Let us study a halting operation for a Turing machine to understand it better.

**Example 3.44.** Given transitions $\delta(q_0, a) = (q_1, a, R)$ and $\delta(q_1, a) = (q_0, a, L)$ for a Turing machine, we will go through the transition as shown in Figure 3.17 and the transition graph shown in Figure 3.18. The machine is in state $q_0$, it reads the input $a$, leaves it as $a$, changes state to $q_1$ and then shifts right. The machine is in state $q_1$ now, it reads the input $a$, leaves it as $a$, changes state to $q_0$ and then shifts left. The cycle continues forever, and the Turing machine never halts. The final state of a Turing machine that doesn’t halt is shown as $\infty$. Hence, for this example to indicate the Turing machine starting from the initial state to no halting as

$$q_0aa \Rightarrow \infty \quad (3.16)$$

For formally we present the language accepted by a given Turing machine as shown below.
**Definition 3.45.** Given a Turing machine \( \mathcal{T} = (Q, \Sigma, \Theta, q_0, \square, F) \), the language accepted by \( \mathcal{T} \) is defined as

\[
\mathcal{L}(\mathcal{T}) = \{ s \in \Sigma^+ | q_0 s \Rightarrow u q_f v, \ q_f \in F, \ u, v \in \Theta^* \} \tag{3.17}
\]

We can build a Turing machine that accepts every language on a given alphabet. It simply moves from the initial state to the final state no matter what the input is. Hence we need classification of better families of languages. For that, we present the next two.

**Definition 3.46.** A language is called **recursively enumerable language** or Turing recognizable language, or semi-decidable if there exists a Turing machine that halts in a final state for every member of this language, and when that Turing machine is given a string not in that language, it either halts in a non-final state (rejecting that string) or never halts. □

**Definition 3.47.** A language is called **recursive** or **decidable** if there exists a Turing machine that halts in a final state for every member of this language, and when that Turing machine is given a string not in that language, it halts in a non-final state (rejecting that string). □

To perform more analysis of these languages we need to review the fundamentals of cardinality (or size) of sets. Fundamentals of set theory are presented in a compact way in [Hal60] and in depth in [Jec03]. On a course in real analysis, cardinality is discussed in an introduction section in [DiB02] like in most books on analysis.

**Definition 3.48.** Two sets \( X \) and \( Y \) have the same **cardinality**, if there is a bijection from \( X \) onto \( Y \), and we say \( \text{card}(X) = \text{card}(Y) \). □

Cardinality of a finite set is the number of elements it contains. If there is an injection from \( X \) to \( Y \) then \( \text{card}(X) \leq \text{card}(Y) \). On the other hand, if there is a surjection from \( X \) to \( Y \), then \( \text{card}(X) \geq \text{card}(Y) \). In particular, \( X \subseteq Y \Rightarrow \text{card}(X) \leq \text{card}(Y) \), and \( X \supseteq Y \Rightarrow \text{card}(X) \geq \text{card}(Y) \).

**Theorem 3.49** (Cantor-Schröder-Bernstein Theorem). \( \text{card}(X) \leq \text{card}(Y) \) and \( \text{card}(X) \geq \text{card}(Y) \) ⇒ \( \text{card}(X) = \text{card}(Y) \) □

**Proof.** Without loss of generality, we will assume that the sets \( X \) and \( Y \) are disjoint. The condition of the theorem tells us that \( \exists f : X \to Y \) and \( \exists g : Y \to X \) which are both injective. Hence, we have

\[
\forall x \in X, \cdots \to f^{-1}(g^{-1}(x)) \to g^{-1}(x) \to x \to f(x) \to g(f(x)) \to \cdots \tag{3.18}
\]
and similarly,
\[ \forall y \in Y, \cdots \rightarrow g^{-1}(f^{-1}(y)) \rightarrow f^{-1}(y) \rightarrow y \rightarrow g(y) \rightarrow f(g(y)) \rightarrow \cdots \] (3.19)

Since both functions \( f \) and \( g \) are injective, they divide \( X \) and \( Y \) into disjoint sets. A sequence can have any of the following three things happen to it.

1. The sequence can terminate in a point, say \( x_0 \) in \( X \) on the left if \( g^{-1}(x_0) \) wouldn’t exist.
2. The sequence can terminate in a point, say \( y_0 \) in \( Y \) on the left if \( f^{-1}(y_0) \) wouldn’t exist.
3. The sequence would never terminate.

Let \( Z_x \) be the set of points in \( X \cup Y \) that stop on the left in \( X \), \( Z_y \) be the set of points in \( X \cup Y \) that stop on the left in \( Y \), and \( Z_z \) be the rest of the points. We define a bijection \( h \) between \( X \) and \( Y \) as follows.

\[ h(x) = \begin{cases} 
  f(x), & \text{if } x \in (Z_x \cup Z_z) \cap X \\
  g^{-1}(f(x)), & \text{otherwise}
\end{cases} \] (3.20)

**Definition 3.50.** A set is **countable** if there is a bijection to it from a subset of natural numbers \( \mathbb{N} \).

Hence, a countable set is either finite, or countably infinite (or **denumerable**).

**Proposition 3.51.** Set of natural numbers \( \mathbb{N} \) is countable.

*Proof.* Take the bijection as \( f(n) \).

**Proposition 3.52.** Set of even natural numbers \( \mathbb{N} \) is countable.

*Proof.* Take the bijection as \( f(2n) \).

**Proposition 3.53.** Set of integers \( \mathbb{Z} \) is countable.

*Proof.* We can put all integers in a sequence as \( \{1, -1, 2, -2, \cdots\} \). The bijective mapping from \( \mathbb{N} \) is then

\[ f(n) = \begin{cases} 
  -n/2, & \text{if } n \text{ is even.} \\
  n + \frac{1}{2}, & \text{otherwise.}
\end{cases} \] (3.21)
Proposition 3.54. Every subset of a countable set is countable.

Proposition 3.55. A set of all finite subsets of a countable set is countable.

Proof. By the fundamental theorem of arithmetic (unique factorization theorem), every natural number has a unique prime decomposition, i.e., it can be written as a product of primes and if the product is written in an ascending order, this factorization is unique. Let the countable set be $E = \{e_1, e_2, \ldots\}$. Then, for any finite subset $F = \{e_{f_1}, e_{f_2}, \ldots, e_{f_k}\}$ of $E$, take the bijection function to be

$$f(F) = 2^{e_{f_1}}3^{e_{f_2}}\cdots p_{e_{f_k}}^{e_{f_k}}$$ (3.22)

where we have taken powers of prime numbers in ascending order. 

Corollary 3.56. Set of pairs of natural numbers is countable.

Corollary 3.57. Set of rational numbers is countable.

Proof. Take the equivalence classes of rational numbers, and for every reduced $p/q$, take $f(p, q) = 2^p3^q$. 

An intuitive and visual way of looking at countability is as follows. Think of any natural number, write it down on a sheet of paper and fold it. Now, if you start counting the natural numbers in a sequence, after a finite amount of time you will reach that number written on that paper, no matter how big that number is. In other words, the entire set can be enumerated in a sequence. The sequence is shown here.

![Figure 3.19: Set of Natural Numbers](image)

Now, let’s see visually the countability of the even numbers. We see that we throw away half the set but the size (cardinality) remains the same. This can not happen in finite sets. If the cardinality of a set is equal to that of its proper subset, then the set can not be finite.

Now, if we look at the set of integers, we see that it goes to infinity on both ends, and it seems strange to see how this can be put in a bijection with $\mathbb{N}$. The solution come by folding the set of integers in the middle (at 0) and bringing the negative side to the positive side. The set of integers is shown in Figure 3.21, and the folded one showing bijection with $\mathbb{N}$ is shown in Figure 3.22.
Figure 3.20: Set of Even Natural Numbers

Figure 3.21: Set of Integers going to Infinity on both Ends

Figure 3.22: Set of Integers showing Bijection

Now, let us look at the set of rational numbers. We can look at it as a sequence of sequences, as \{1/1, 1/2, 1/3, \ldots\}, followed by \{2/1, 2/2, 2/3, \ldots\}, etc. These countable sequence of countable sets also has the same cardinality as \(\mathbb{N}\), as can be visualized by putting all of the items in a single sequence travelling diagonally as seen in Figure 3.23.

Now, if we add a zero, or all negative rationals as well, we can easily count alternately as before and obtain a single sequence.

Now, we present one of the most important and famous propositions in mathematics which uses Cantor’s diagonalization method.

**Proposition 3.58** (Cantor). Interval \([0, 1]\) of real numbers is not countable.

*Proof.* To visualize the proof, please refer to Figure 3.24. We do this proof by contradiction. Assume that all real numbers between 0 and 1 can be placed in a sequence. Each number has a decimal (infinite digit) representation. The first number is \(0.a_{11}a_{12}a_{13}\ldots\), the second one is \(0.a_{21}a_{22}a_{23}\ldots\), etc. Now from this sequence of real numbers we create the following real number \(x = 0.r_1r_2r_3\ldots\), as follows.

\[
r_k = \begin{cases} 
1 & \text{if } a_{jj} \text{ is even} \\
2 & \text{if } a_{jj} \text{ is odd}
\end{cases}
\]  

(3.23)
The number $x \in [0,1]$ and can not be equal to any of the numbers in the sequence because it is different from the diagonal members of the sequence as shown in Figure 3.24, and hence is different from every number in at least one digit.

Theorem 3.59 (Cantor’s Theorem). \textit{For any set $X$,} $\text{card}(2^X) > \text{card}(X)$. 

\[\square\]
Proof. In order to prove this theorem, we show that no function from any set \( X \) to its power set \( 2^X \) can be surjective. For this, we will produce a subset of \( X \), i.e. a set \( Y \in 2^X \) which will not be an image for any \( x \in X \). That set is

\[
Y = \{ x \in X | x \not\in f(x) \} \tag{3.24}
\]

For any \( x \in X \), either \( x \in f(x) \) or \( x \not\in f(x) \). We show that in both cases \( f(x) \neq Y \) hence, proving the theorem. If \( x \in f(x) \), then \( x \not\in Y \) by the definition of \( Y \) and therefore, \( f(x) \neq Y \). On the other hand, if \( x \not\in f(x) \), then \( x \in Y \), and therefore, again, \( f(x) \neq Y \).

3.4.2 Turing Decidable and Semi-decidable Languages

Proposition 3.60. There is an enumeration procedure for any Turing decidable language.

Proof. Turing decidable language (recursive language) has a Turing machine such that given any string of the language alphabet, it halts in the final state if the string belongs to that language otherwise rejects it. This enables an easy enumeration procedure for this language.

Given the alphabet \( \Sigma \), all the strings of \( \Sigma^* \) can be easily enumerated. For each string the Turing machine accepts or rejects that string. Hence, as the strings that are accepted are obtained, they make an enumeration of the language.

Proposition 3.61. There is an enumeration procedure for any Turing semi-decidable language.

Proof. Turing semi-decidable language (recursively enumerable language) has a Turing machine such that given any string of the language, it halts in the final state. However, if the string does not belong to the language then the Turing machine might not halt.

Given the alphabet \( \Sigma \), all the strings of \( \Sigma^* \) can be enumerated. If a string belongs to the language, the Turing machines will accept it in a finite number of steps. However, if the string doesn't belong to the language, the machine would be stuck. However, a diagonal technique allows the enumeration of this language as well as show in Figure 3.25. The machine does not work on one string till it decides that string. Instead it works on different strings simultaneously one at a time diagonally as shown in Figure 3.25, and this way, all the strings from the language that it decides in finite number of steps get enumerated one by one.
3.4. TURING MACHINES AND LANGUAGES

**Theorem 3.62.** There are countably many Turing machines.

*Proof.* Each Turing machine can be coded with symbols. Without loss of generality, let us assume that we perform the coding using the alphabet \( \{0, 1\} \). The set \( \{0, 1\}^* \) is countable, and a subset of that will represent Turing machines. A subset of a countable set is countable.

**Theorem 3.63.** There is a language that is not Turing semidecidable.

*Proof.* Take \( \{0, 1\} \) as the alphabet. The set of all strings from this alphabet is \( \{0, 1\}^* \) which is countable. The power set of this is the set of all languages on this alphabet. The cardinality of this power set is \( 2^\mathbb{N} \). On the other hand the number of Turing machines and consequently Turing semidecidable languages is countable, and hence there is a language which is not Turing semidecidable.

**Theorem 3.64.** There is a language that is Turing semidecidable but not Turing decidable.

*Proof.* The statement means that there is a language \( \mathcal{L} \) which is Turing semidecidable but its complement \( \mathcal{L}^c \) is not Turing semidecidable, because if \( \mathcal{L}^c \) were also Turing semidecidable, then \( \mathcal{L} \) would be Turing decidable.

Let us fix an alphabet \( \Sigma \) and enumerate all its strings as \( s_1, s_2, \text{ etc.} \). Let us also use \( T_1, T_2, \text{ etc.} \) as the enumeration of all the Turing machines. We will call Turing semidecidable language associated with a Turing machine \( T_i \) to be \( \mathcal{L}(T_i) \). Define \( \mathcal{L} \) as:

![Figure 3.25: Turing Semidecidable Language Enumeration](image-url)
\( \mathcal{L} = \{ s_i s_i \in \mathcal{L}(T_i) \} \) \hspace{1cm} (3.25)

The complement of this language is

\( \mathcal{L}^c = \{ s_i s_i \not\in \mathcal{L}(T_i) \} \) \hspace{1cm} (3.26)

Language \( \mathcal{L} \) is Turing semidecidable because we can run (diagonally, just like we did in Figure 3.25) all Turing machines so that they generate their strings one at a time, and we can check if the string generated from each \( T_i \) matches \( s_i \). If it does, it becomes a member of \( \mathcal{L} \). This becomes an enumeration scheme for this language.

Now, we will show that the language \( \mathcal{L}^c \) is not semidecidable, i.e. there is no \( T_k \) such that \( \mathcal{L}^c = \mathcal{L}(T_k) \). We will prove this by contradiction. Let us assume that such a \( T_k \) exists. If it does, we ask, does \( s_k \in \mathcal{L} \). There are two possibilities:

1. If \( s_k \in \mathcal{L} \), then \( s_k \in \mathcal{L}(T_k) \) by the definition of the language \( \mathcal{L} \). However, since \( T_k \) is the Turing machine that accepts strings of \( \mathcal{L}^c \), i.e. those strings that \( \not\in \mathcal{L} \), this creates a contradiction.
2. If \( s_k \not\in \mathcal{L} \), then it would mean that \( s_k \in \mathcal{L}(T_k) \), and which would imply that \( s_k \in \mathcal{L} \) giving contradiction again.

Hence, the only possibility is that no such \( T_k \) exists, and therefore, \( \mathcal{L}^c \) is not Turing semidecidable.

Unrestricted Grammar and Turing Semidecidable Languages

**Definition 3.65.** An **unrestricted grammar** is a quadruple \((N, T, S, \mathcal{P})\), where the production rules are of the form given below, where \( x \in (N \cup T)^+ \) and \( x \in (N \cup T)^* \).

\[
\begin{array}{c}
\text{Substitution} \\
x \ ::= \ y
\end{array}
\]
Theorem 3.66. Every unrestricted grammar generates a Turing semidecidable language, and inversely, to every Turing semidecidable language, there exists an unrestricted grammar that generates the same language.

Proof. Following the sequential steps of the grammar, we can enumerate all the strings of the language. An unrestricted grammar can be designed that stepwise simulates all the steps of a Turing machine and hence creates a language enumerated by that machine. The details of this transformation are presented in [Lin11].

Context Sensitive Grammar and Linear Bounded Automata

Definition 3.67. A context sensitive grammar is a quadruple \((N, T, S, P)\), where the production rules are of the form given below, where \(x, y \in (N \cup T)^+\), and the length of the string \(x\) has to be less than or equal to that of \(y\). In other words, the productions are non-contracting.

<table>
<thead>
<tr>
<th>Table 3.10: Context Sensitive Grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Substitution</td>
</tr>
<tr>
<td>(x ::= y)</td>
</tr>
</tbody>
</table>

Definition 3.68. A linear bounded automata is a nondeterministic Turing machine that can during its operations, only use the input cells on the tape on which the input string is placed.

Theorem 3.69. Every \(\lambda\)-free context free language has a corresponding linear bounded automata that accepts the exact same language. Inversely, for every linear bounded automata, there exists a context free grammar generating the same language.

Proof. Since a linear bounded automata is nondeterministic, it can perform the derivation of the string, and since the derivations are non-contracting, those will not require more than the length of the input. The constructive proof of the inverse is similar to the one for the inverse part of Theorem 3.66.
Every context sensitive language is Turing semidecidable, since a linear bounded automata is a Turing machine. These languages are also recursive (Turing decidable), because their productions are noncontracting and hence their membership is decidable in finite number of production steps. To show a language which is recursive or Turing decidable but not context sensitive, the diagonalization construction very similar to that of Theorem 3.64 can be used (see [Lin11] for details).

### 3.5 Chomsky Hierarchy and Language Containments

In summary, we have seen various types of languages, grammars and machines. We went from languages that were regular to context free, to context sensitive, to Turing decidable (recursive) to Turing semidecidable (recursively enumerable). We had associated grammars for regular (right linear and left linear), context free, context sensitive, and unrestricted grammar for Turing semidecidable language. The corresponding acceptors were finite state machines for regular, nondeterministic pushdown automata for context free, linear bounded automata for context sensitive, and Turing machines for Turing decidable and semidecidable.

Comparing the grammars, the Chomsky hierarchy calls the languages generated by unrestricted grammars type-0, those generated by context sensitive grammars type-1, context free type-2, and finally regular as type-3. Their containment is as follows:

\[
\text{type-3} \subset \text{type-2} \subset \text{type-1} \subset \text{type-0} \quad (3.27)
\]

We can add the other languages we studied too, and give the following relationships.

\[
\mathcal{L}_{\text{reg}} \subset \mathcal{L}_{\text{dcf}} \subset \mathcal{L}_{\text{cf}} \subset \mathcal{L}_{\text{cs}} \subset \mathcal{L}_{\text{Td}} \subset \mathcal{L}_{\text{Tsd}} \quad (3.28)
\]

where, \( \mathcal{L}_{\text{reg}} \) meant regular language, \( \mathcal{L}_{\text{dcf}} \) deterministic context free, \( \mathcal{L}_{\text{cf}} \) context free, \( \mathcal{L}_{\text{cs}} \) context sensitive, \( \mathcal{L}_{\text{Td}} \) Turing decidable, and \( \mathcal{L}_{\text{Tsd}} \) Turing semidecidable.

When we compare the machines, we see that the finite state machines had finite storage. We added infinite stack to a pushdown automata. In fact if we make the stack finite for a pushdown automata, it in fact becomes equivalent
to a finite state machine. Linear bounded automata were restricted Turing machines, and then finally the Turing machine.

We can summarize the relationship between languages off the Chomsky hierarchy, their grammars and machines that accept those languages as shown in Table 3.11.

<table>
<thead>
<tr>
<th>Language</th>
<th>Grammar</th>
<th>Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular Language</td>
<td>Left, or Right Linear</td>
<td>Finite State</td>
</tr>
<tr>
<td>Context-free</td>
<td>Context-free</td>
<td>Pushdown Automata</td>
</tr>
<tr>
<td>Context-sensitive</td>
<td>Context-sensitive</td>
<td>Linear Bounded Automata</td>
</tr>
<tr>
<td>Recursively Enumerable</td>
<td>Unrestricted</td>
<td>Turing Machine</td>
</tr>
</tbody>
</table>

Table 3.11: Grammar for Example 3.4

3.6 Bibliographical Notes

We discuss bibliography related to this section ···

3.7 Exercises

Problem 3.1. Design a grammar that generates the language $L = \{a^b n, n \geq 0\}$. Notice that this case is different from the Example 3.4 because the string $b$ does not belong to the language generated by that grammar, but it must be included in the language generated by the grammar of this problem.

Problem 3.2. Prove that for any $k \in \mathbb{N}$, $\text{card}(\mathbb{N}^k) = \text{card}(\mathbb{N})$, and also that $\text{card}(2^{\mathbb{N}}) = \text{card}(\mathbb{R})$.

Problem 3.3. Prove that for any $\text{card}((\{0, 1\}^\mathbb{N}) = \text{card}(\mathbb{R})$.

Problem 3.4. Prove that $\text{card}(2^\mathbb{N} \times 2^\mathbb{N}) = \text{card}(2^\mathbb{N}) = \text{card}(\mathbb{R})$.

Problem 3.5. Prove that for any sets $X, Y, Z, \text{card}(X^Y \times Z) = \text{card}((X^Y)^Z)$.

Problem 3.6. Prove that for any $k \in \mathbb{N}$, $\text{card}(\mathbb{R}^k) = \text{card}(\mathbb{R})$.

Problem 3.7. Prove that $\text{card}(\mathbb{N}^\mathbb{N}) = \text{card}(\mathbb{R})$. 
This chapter is devoted to presenting the complexity theory of algorithms in order to explain what P and NP class of problems are, and why it is important to understand which algorithm falls in which category. In order to explain the two and other categories of algorithms, this chapter will first review computability and decidability and then study the various time and space complexity classes of problems.

4.1 Equivalent Turing Machines

We have already formally defined Turing machines in Section 3.4.1 of Chapter 3. Now, we will show Turing machines with different features, but noticing that they are all equivalent to the standard Turing machine, in the sense that for each of these modified Turing machines, the standard Turing machine can simulate that Turing machine, which means it can accept the same language as that Turing machine, and vice versa.

4.1.1 Turing Machine with Stay

Turing machine with a stay option has a third option for its read-write head. In addition to moving right and left, it can also just stay at the same cell, i.e. the options are \{L, R, S\}. A Turing machine with stay option can easily simulate a standard Turing machine by not using the S option. On the other hand, a standard Turing machine can also simulate the one with the stay option,
by making a loop to come back to the same cell, for instance by moving to the right, and then back to the left. For instance, \( \delta(q_0, a) = (q_1, b, S) \) can be replaced by \( \delta(q_0, a) = (q_1, b, R) \) and \( \delta(q_1, a) = (q_1, a, L) \), if there is an \( a \) in the cell on the right.

### 4.1.2 Multitrack Turing Machine

A multitrack Turing machine uses multiple cells instead of one cell on a tape, as shown in Figure 4.1. For the example shown in Figure 4.1, the number of cells in parallel used are three. The equivalence of a multitrack Turing machine with a standard Turing machine is easily seen when we equate each possible \( n \)-symbol string of the \( n \)-track multitrack Turing machine with a single symbol of the standard Turing machine. For instance, if there are two cells and two symbols \( \{a, b\} \) on the two tracks, they would equate to four different strings \( \{aa, ab, ba, bb\} \) for the single track machine. We can view them as four symbols \( \{a_1, a_2, a_3, a_4\} \).

![Figure 4.1: Multitrack Turing Machine](image)

### 4.1.3 One-way Tape Turing Machine

A standard Turing machine is equivalent to a two-track one-way Turing machine. This can be shown by folding a one track two-way Turing machine at the middle, and then viewing it as a two-track one-way tape. The transitions of the two-track one-way machine are designed to make them equivalent to that of the single track. Then, finally, two-track machine can be transformed to one-track as shown before.

### 4.1.4 Multi-Tape Turing Machine

A multi-tape Turing machine as shown in Figure 4.3, where we show a two-tape one, has one state corresponding to the current cell on multiple tapes,
4.1. EQUIVALENT TURING MACHINES

where the transition causes a simultaneous write operations and move operations on those tapes in parallel, and the state of the system changes. Like in the other cases, multi-tape Turing machine is also completely equivalent to a standard Turing machine.

4.1.5 Read-only Tape (Off-line) Turing Machine

Turing machine with a read only tape has this additional tape to read data. This data is read, and not written to. The transition function of this machine maps the symbol read from this tape, symbol read from the read-write tape, and the current state of the machine to its new state, the symbol written on the read-write tape, and the motion on the read-write tape.

4.1.6 Multidimensional Turing Machine

A two-dimensional Turing machine has a tape that extends to infinity in all four directions, as shown in Figure 4.5. An $n$-dimensional Turing machine has a tape that extends in all the $2^n$ directions to infinity.

4.1.7 Nondeterministic Turing Machine

A nondeterministic Turing machine is just like a Turing machine except that its transition function is not the one used for deterministic Turing machine as
shown in Equation 4.1.

\[ \delta : \Omega \subset Q \times \Theta \rightarrow Q \times \Theta \times \{L \times R\} \] (4.1)

The transition function for a nondeterministic Turing machine, instead is the one as shown in Equation 4.2.

\[ \delta : \Omega \subset Q \times \Theta \rightarrow 2^{Q \times \Theta \times \{L \times R\}} \] (4.2)

What this means is that a nondeterministic Turing machine has set valued transitions. It can transition to multiple states from a single state and a single input. Hence it can have multiple arcs on its transition graph for the same input. An example of this is shown in the transition graph of a nondeterministic Turing machine in Figure 4.6.
4.2. DECIDABILITY AND COMPUTABILITY

![Diagram of a nondeterministic Turing Machine]

Every deterministic Turing machine is also a nondeterministic Turing machine, and inversely, a deterministic Turing machine can be designed for a given nondeterministic Turing machine to simulate it by making it transition through each of the branches.

4.1.8 Universal Turing Machine

Just like a computer that can be designed to do a specific task (as an embedded system), and it can also be used as a general purpose computer which runs other programs, we can have a Turing machine that does a specific task, as we have seen till now, or a general purpose Turing machine, which we call universal Turing machine, that, given the description of another Turing machine and an input for that Turing machine simulates that Turing machine.

4.2 Decidability and Computability

Decidability is the topic of studying decision problems and effective solutions for them.

**Definition 4.1** (Decision Problem). A decision problem is a doublet \( \{P, \mathcal{E}\} \), where:

- **Set** \( P \) : is the set of problems, and
- **Evaluation Function** \( \mathcal{E} \) : is a function \( \mathcal{E} : P \rightarrow \{0, 1\} \).

The co-domain of the evaluation function is \( \{0, 1\} \), which can be interpreted as false, true, or no, yes answer to every question in the problem set \( P \).

An **algorithm** \( \mathcal{A} \) to solve the decision problem, informally is a procedure that also provides mechanistically the mapping \( \mathcal{E} : P \rightarrow \{0, 1\} \) such that

\[
\forall p \in P, \mathcal{A}(p) = \mathcal{E}(p)
\] (4.3)
This means that the algorithm should solve the problem, i.e., it gives the correct answer. An effective algorithm has the following characteristics:
1. It should solve the problem correctly in its domain, and
2. It should provide the solution in a finite number of steps for each \( p \in P \).

Turing machines can be designed to perform operations such as addition, multiplication, exponentiation, conditional computation (such as if-then-else), loops etc. In fact, anything a computer can do, or any mechanistic device can do a Turing machine do. Based on this observation and experience, we define algorithm in terms of Turing machine operations as shown in the two thesis statements below.

**Proposition 4.2** (Church Turing thesis). An effective algorithm to solve a decision problem exists, or in other words, the problem is decidable, if and only if there exists a Turing machine that halts on every \( p \in P \) and answers correctly.

**Proposition 4.3** (Extended Church Turing thesis). An effective algorithm to partially solve a decision problem exists, or in other words, the problem is semidecidable, if and only if there exists a Turing machine that halts on every \( p \in P \) for which the answer is 1 or yes and answers correctly every time it halts.

The decision problem can be easily converted into a language membership problem, as follows:

**Definition 4.4** (Language Membership Problem). A Language Membership Problem is a triplet \( \{ \Sigma, \mathcal{L}, \omega \} \), where:
- **Set \( \Sigma \)**: is a given finite alphabet,
- **Set \( \mathcal{L} \)**: is a given language over \( \Sigma \), and
- **String \( \omega \)**: is a string over \( \Sigma \), i.e. \( \omega \in \Sigma^* \).

We have already seen the following two results in Chapter 3 in Definition 3.46 and Definition 3.47.

**Proposition 4.5.** A language membership problem is decidable if \( \mathcal{L} \) is recursive.

**Proposition 4.6.** A language membership problem is semidecidable if \( \mathcal{L} \) is recursively enumerable.

### 4.2.1 Computability

When we view a Turing machine as a transducer, we see that it transforms a given input string into an output string if it halts. Hence a Turing machine would represent a function if on the domain of the function it would halt at
every input and consequently provide an output string for every input. We can say that as $q_0s_i \Rightarrow q_f s_0$, which shows that in the initial configuration the input string $s_i$ is on the tape with the read-write head to the left of it, and the Turing machine halts in the final state with its head on the left of the output string. Using this, we define a computable function formally as

**Definition 4.7** (Computable Function). A function is called **computable** if

$$ \exists \Sigma = (Q, \Sigma, \Theta, \delta, q_0, \square, F), $$

such that

$$ q_0 u \Rightarrow_{\Sigma} q_f f(u), \quad q_f \in F, \quad u, f(u) \in \Theta^* $$

(4.4)

It is assumed that $u \in \text{domain of the function } f$.

Every Turing machine represents a function. However, since a Turing machine might halt on some input, we define two types of functions similar to how we defined languages relating to the halting and non-halting behavior of Turing machines. In the case of languages we had a language that is recursive, if a Turing machine halts on every input, and halts on a final state for every string from that language. That same Turing machine creates an output string for every input string of that language. That function is the computable function.

Now, the language that is recursively enumerable shows the case when a Turing machine accepts the strings of a language, but on the other strings it can either halt in a non-final state, or never halt. We define two different types of functions based on these observations.

**Definition 4.8** (Total Function). A function $f : X \supset D \rightarrow Y$ is a relation which is called a **total function** if

1. **Unique Value**: $(x, y_1) \in f, (x, y_2) \in f \Rightarrow y_1 = y_2$, and
2. **Total Property**: $\forall x \in X, \exists y \in Y \ (x, y) \in f$. In other words, $D = X$.

**Definition 4.9** (Partial Function). A function $f : X \supset D \rightarrow Y$ is a relation which is called a **partial function** if

- **Unique Value**: $(x, y_1) \in f, (x, y_2) \in f \Rightarrow y_1 = y_2$

Just like we proved that there is a language that is not recursively enumerable, we can prove using Cantor’s diagonalization, that there is a function that is not computable (see [Dav82]).

**Theorem 4.10** (Noncomputable Function). There is a function $f : \mathbb{N} \rightarrow \mathbb{N}$ that is not computable.
Proof. Since each computable function is associated with a Turing machine, and there are countable number of Turing machines, there are countable number of computable functions \( f_i : \mathbb{N} \to \mathbb{N}, i \in \mathbb{N} \). Define a new function \( g : \mathbb{N} \to \mathbb{N} \) as

\[
g(i) = f_i(i) + 1 \quad (4.5)
\]

It is clear that \( g : \mathbb{N} \to \mathbb{N} \) is a well defined function, and it is different than every computable function \( f_i(k) \) at \( k = i \), and hence is not computable. \( \square \)

Some Computable Functions

Here we provide some examples of computable and partially computably functions from \( \mathbb{N} \to \mathbb{N} \). We will not provide explicit Turing machines that compute these functions. The readers can refer to [Dav82] to get details of those Turing machines that accomplish this task. As an example of the details though, for the addition function, the input functions are placed on the tape as \( n \) number of symbol 1 to represent the natural number \( n \). Placing \( n \) and \( m \) number of symbol 1 on both sides of a blank symbol gives the input on the tape. The transition function of the Turing machine is designed to produce \( n + m \) number of symbol 1 on the tape in the final state.

**Example 4.11** (Zero Function). Design a Turing machine such that \( f(x) = 0 \). \( \triangle \)

**Example 4.12** (Addition). Design a Turing machine such that \( f(x, y) = x + y \). \( \triangle \)

**Example 4.13** (Successor). Design a Turing machine such that \( S(x) = x + 1 \). \( \triangle \)

**Example 4.14** (Partial Subtraction). Design a Turing machine such that \( f(x) = x - y \) if \( x \geq y \). \( \triangle \)

**Example 4.15** (Full Subtraction). Design a Turing machine such that

\[
f(x) = \begin{cases} 
  x - y, & \text{if } x \geq y \\
  0, & \text{otherwise}
\end{cases} \quad (4.6)
\]

\( \triangle \)

**Example 4.16** (Identity). Design a Turing machine such that \( f(x) = x \). \( \triangle \)
4.2. DECIDABILITY AND COMPUTABILITY

Example 4.17 (Projection). Design a Turing machine such that
\( P^n_i(x_1, x_2, \ldots, x_n) = x_i. \)

Example 4.18 (Multiplication). Design a Turing machine such that \( f(x, y) = xy. \)

Example 4.19 (Composition). Design a Turing machine such that \( f(x) = h(g_1(x), g_2(x), \ldots, g_m(x)), \) where \( h \) and \( g_i \) are computable functions, \( h \) is an \( m \)-ary function, and \( x \in \mathbb{N}^n. \)

Example 4.20 (Recursion). Design a Turing machine such that for a given computable function \( g \) and \( h, x \in \mathbb{N}^n, g : \mathbb{N}^n \to \mathbb{N}, h : \mathbb{N}^{n+2} \to \mathbb{N}, \) we have
\[
\begin{align*}
    f(x, 0) &= g(x) \quad (4.7) \\
    f(x, y + 1) &= g(h(x, y, f(x, y))) \quad (4.8)
\end{align*}
\]

We can also go another way, and define a class of functions from primitive functions.

Definition 4.21 (Primitive Recursive Functions). A function is called primitive recursive if it is either the zero function, a successor function, or a projection function, or obtained by successive composition or recursion on a primitive recursive function.

We can prove easily that the functions like addition, subtraction, multiplication, factorial, etc. are primitive recursive.

Definition 4.22 (Ackerman’s Function). Ackerman’s function is given by:
\[
\begin{align*}
    A(0, y) &= y + 1 \\
    A(x, y) &= A(x - 1, 1) \\
    A(x, y + 1) &= A(x - 1, A(x, y))
\end{align*}
\]

It can be proven by studying the growth of the functions (see [DDQ78]) that the Ackerman’s function is not primitive recursive.

Definition 4.23 (Minimalization). Given a total function \( g : \mathbb{N}^{n+1} \to \mathbb{N}, \) then we can produce a new function \( f : \mathbb{N}^n \to \mathbb{N} \) from \( g \) by applying the \( \mu \) operator as follows
\[
f(x) = \mu y [g(x, y) = 0]
\]
where $\mu y[g(x, y) = 0]$ is the least natural number such that $g(x, y) = 0$, and where $x \in \mathbb{N}^n$.

**Definition 4.24** ($\mu$-Recursive Functions). A function is called $\mu$-recursive if it is either the zero function, a successor function, or a projection function, or obtained by successive composition, recursion, or minimalization on a $\mu$-recursive function.

**Theorem 4.25.** A function is computable if and only if it is $\mu$-recursive.

It is important to realize that in this theorem, we must consider only total functions, and not partial functions. The application of $\mu$ to partial functions can create a non-computable function.

### 4.2.2 Decidable Problems

We have seen some decidable problems for regular and context-free languages. We re-state them here for convenience.

**Theorem 4.26.** Given a grammar for a regular language $L$ on an alphabet $\mathcal{A}$ and a string $\omega \in \mathcal{A}^*$, then an algorithm in a finite number of steps can ascertain if $\omega \in L$ is true or false.

**Theorem 4.27.** It is decidable if a given regular grammar has a language that is empty, finite, or infinite.

**Theorem 4.28.** It is decidable if two given regular languages are equal.

**Theorem 4.29.** Given a context free grammar, it is decidable if its corresponding language is empty.

**Theorem 4.30.** Given a context free grammar, it is decidable if its corresponding language is infinite.

**Theorem 4.31.** Given a context free grammar and a string, it is decidable whether the string belongs to the language or not.

### 4.2.3 Halting Problem

The halting problem is stated as follows.

**Definition 4.32** (Halting Problem). Will a Turing machine $\mathcal{T}$ given an input string $\omega \in \Sigma^*$ halt?
We can give a definition of the halting problem in terms of a Turing machine answering this question. A Turing machine can be encoded using 0 and 1 which describes its transitions etc. (see [HMU07] for more details and example). This representation of a Turing machine can be given to another Turing machine with a string \( \omega \) so that this Turing machine can try to decide if the Turing machine whose string description is given will halt or not if it executes the given string. This definition is formalized below (see [Lin11]).

**Definition 4.33 (Halting Problem).** Given a Turing machine description \( \omega \) and a string \( \omega \), does there exist a Turing machine \( \Sigma_H \), which we will call the **Halting Oracle Turing Machine**, which has its initial state \( q_0 \) and two distinct final states \( q_T \) and \( q_F \) such that if the Turing machine \( \Sigma \) halts with \( \omega \) as its input, then the Turing machine \( \Sigma_H \) follows:

\[
q_0 \omega \Sigma \omega \Rightarrow_{\Sigma_H} s_1 q_T s_2
\]

and if the Turing machine \( \Sigma \) does not halt with \( \omega \) as its input, then the Turing machine \( \Sigma_H \) follows:

\[
q_0 \omega \Sigma \omega \Rightarrow_{\Sigma_H} s_3 q_F s_4
\]

We present the fundamental theorem of decidability.

**Theorem 4.34 (Fundamental Theorem of Decidability).** The halting problem is undecidable.

*Proof 1:* The proof of theorem is a simple consequence of a theorem we already saw in Chapter 3, which was Theorem 3.64. That theorem stated that there is a recursively enumerable (Turing semidecidable) language that is not recursive (Turing decidable). To prove the fundamental theorem now, let us assume that the Halting Oracle TM exists, which, given a string representation of any Turing machine and any string over the alphabet of that input string Turing machine, halts for every such input to answer the halting question for that Turing machine that has been coded as a string in its input. This would mean that, if we have any recursively enumerable language, we would take the Turing machine corresponding to that language, and ask the halting question to the Halting Oracle TM giving this Turing machine as an input and any string of this Turing machine's language. We would get an answer every time, and that would make this language recursive. Since, that is not true, hence Halting Oracle TM does not exist.
Another proof we will provide next depends on the relationship to computable and partially computable functions, as compared to the proof 1, that depended on the relationship to languages.

**Proof 2:** Assume that the Halting Oracle TM exists. Since there are a countable number of Turing machines, we have a bijection from $\mathbb{N}$ to the set of all Turing machines. Hence, we can enumerate all Turing machines, and use a natural number to indicate which Turing machine we are referring to. Similarly, the total number of finite strings from all set of finite alphabet is a countable collection of countable objects, and hence is countable itself. Therefore, we can also give a unique natural number to every string as well. Hence, the Halting Oracle TM can be considered as a function that given a natural number representing a Turing machine, and another natural number representing a given string, outputs a 1 if the given Turing machine will halt given that string as an input, and outputs a 0 otherwise. We show this function as:

$$H_T(i, j) = \begin{cases} 
1, & \text{if Turing Machine } i \text{ halts on string } j \\
0, & \text{otherwise}
\end{cases} \tag{4.11}$$

Take any computable binary function $f(i, j)$ (i.e., with $\{0, 1\}$ as the domain) and based on that create a new binary function $g(i)$ as follows:

$$g(i) = \begin{cases} 
0, & \text{if } f(i, i) = 0 \\
u, & \text{otherwise}
\end{cases} \tag{4.12}$$

Here, $u$ means undefined. Now the partial function $g(\cdot) : \mathbb{N} \rightarrow \{0, 1\}$ is partially computable, since we have an algorithm to compute its value from those of $f(i, i)$ function. Hence, there exists a natural number $k$ such that the Turing machine that $k$ represents will correspond to the partially computable function $g(\cdot)$. Now, let’s compare $H_T(k, k)$ with $f(k, k)$. There are the following two possibilities:

1. $g(k) = 0$, which implies that $f(k, k) = 0$, and since this means that the Turing machine halted $H_T(k, k) = 1$, or
2. $g(k) = u$, which implies that $f(k, k) \neq 0$, and since this means that the Turing machine does not halt, and therefore $H_T(k, k) = 0$.

Hence, we have proven that $H_T$ can not be equal to any computable function $f$. \hfill \Box

Now, we provide a proof that directly builds a Turing machine to provide the proof.
Proof 3: Assume the existence of the Halting Oracle TM.

\[
H_T(i, j) = \begin{cases} 
1, & \text{if Turing Machine } i \text{ halts on string } j \\
0, & \text{otherwise} 
\end{cases} \quad (4.13)
\]

We apply the Turing machine \( i \) to string \( j \), and have the following Turing machine.

\[
H_M(i) = \begin{cases} 
1, & \text{if Turing Machine } H_T \text{ halts} \\
0, & \text{otherwise} 
\end{cases} \quad (4.14)
\]

We build a new Turing machine from this one that produces the following behavior.

\[
H_c(i) = \begin{cases} 
\text{loop infinitely,} & \text{if Turing Machine } H_M(i) = 1 \\
\text{halt,} & H_M(i) = 1 
\end{cases} \quad (4.15)
\]

Now, let us take \( H_c \) to indicate the natural number that corresponds to the natural number for the Turing machine \( H_c \) in the argument of \( H_c \) and apply \( H_c \) to itself, i.e. we try to evaluate \( H_c(H_c) \). There are two possibilities:

1. \( H_M(i) = 1 \), which means that \( H_c \) halts, and therefore, now from the description of \( H_c \), it should not halt, or

2. \( H_M(i) = 0 \), which means that \( H_c \) does not halt, and therefore, now from the description of \( H_c \), it should halt.

Since both possible conditions lead to contradiction, the assumption of the existence of the Halting Oracle TM must be false. \( \square \)

### 4.2.4 Undecidable Problems via Reducibility

In order to study decidability of many problems we can study them by equating those to other problems whose result is known. The same technique was used in proof-1 of the halting problem, where it was related to the result on languages.

**Theorem 4.35** (Blank Tape Halting Problem). *Whether any Turing machine that starts on a blank tape will halt or not is undecidable.* \( \square \)

**Proof.** If the blank tape halting problem were decidable, then given any Turing machine \( \Sigma \) and a string \( \omega \), we could find a Turing machine that starts on
a blank tape, writes \( \omega \) on it and then finds the answer to its own halting question. Hence, decidability of the blank tape halting problem would imply that of the halting problem. Hence, the blank tape halting problem is also undecidable. This reduction of the halting problem to the blank tape halting problem is shown in Figure 4.7.

**Theorem 4.36** (State Entry Problem). Whether given any Turing machine will enter a given state or not is undecidable.

*Proof.* If the state entry problem were decidable, then given any Turing machine \( \Sigma \) and a string \( \omega \), we could define a Turing machine that would enter a final state only if it halts. Since, a Turing machine halts if it has an undefined transition, we can complete those transitions to make the state go to this final state. Hence, decidability of the state entry problem would imply that of the halting problem. Hence, the state entry problem is also undecidable. This reduction of the halting problem to the state entry problem is shown in Figure 4.8. In this figure, we show \( \Sigma_m \) as the modified Turing machine.

**Theorem 4.37** (Membership Problem). Whether given any Turing machine and a string will the Turing machine accept the string is undecidable. In other
words to find if a given string belongs to a language generated by an unrestricted grammar is undecidable.

**Proof.** If the membership problem were decidable, then given any Turing machine $\Sigma$ and a string $\omega$, we could define a Turing machine that would enter an accepting final state only if it halts. We could accomplish this by making all undefined transitions go to the accepting final state and also make all the un-accepting final states go to the accepting ones. Hence, decidability of the membership problem would imply that of the halting problem. Hence, the membership problem is also undecidable. This reduction of the halting problem to the membership problem is shown in Figure 4.9. In the figure, we show $L(\Sigma_m)$ to show the language related to the unrestricted grammar related to the given Turing machine, or in other words, the language of the strings accepted by the Turing machine.

![Figure 4.9: Reduction to Membership Problem](image)

**Theorem 4.38** (Empty Language Problem). Whether given any Turing machine, will the Turing machine accept any string is undecidable. In other words to find if a given language generated by an unrestricted grammar is empty is undecidable.

**Proof.** If the empty language problem were decidable, then given any Turing machine $\Sigma$ and a string $\omega$, we could define a new Turing machine that would behave in the following way. If the new Turing machine is given any string and the old Turing machine would accept it, i.e. would reach a final state, then the new Turing machine would accept that string only if it were $\omega$. This can be accomplished by modifying the original Turing machine to save $\omega$ on some part of the input tape, and then modify the transitions so that when it reaches final state, it compares the input string with $\omega$, and accepts if those two are the same. This means that the new Turing machine accepts either only $\omega$ or
no other string. Hence, if we would ask the empty language oracle Turing machine this question, we would find out if \( \omega \) would be accepted or not. Hence, decidability of the empty language problem would imply that of the membership problem. Hence, the empty language problem is also undecidable. This reduction of the membership problem to the empty language problem is shown in Figure 4.10.

\[ \begin{align*}
\text{Membership Oracle TM} & \\
T, \omega & \xrightarrow{\text{Create } \Sigma_m} \Sigma_m & \text{Empty Language Oracle TM} \\
& \xrightarrow{\omega \in \mathcal{L}(\Sigma)} \mathcal{L}(\Sigma_m) \neq \phi \\
& \xrightarrow{\omega \notin \mathcal{L}(\Sigma)} \mathcal{L}(\Sigma_m) = \phi
\end{align*} \]

Figure 4.10: Reduction to Empty Language Problem

In fact, any non-trivial property is undecidable for Turing machines. This can be stated in terms of recursively enumerable languages, and this statement is called Rice’s theorem. In order to set it up, we will provide some definitions (see [HMU07]). Let us consider the alphabet \( \mathcal{A} = \{0, 1\} \). Let \( \ell_{re} \) be some set of recursively enumerable languages over \( \mathcal{A} \). \( \ell_{re} \) is called a property of recursively enumerable languages, and we say that \( L \) has this property if \( L \in \ell_{re} \). Some example of properties are languages being infinite, finite, regular, etc. A property \( \ell_{re} \) is called trivial if it is the empty set, or contains all recursively enumerable languages.

**Theorem 4.39 (Rice’s Theorem).** Any nontrivial property of recursively enumerable languages is undecidable.

**Proof.** To reduce the halting problem to the decidability of a given nontrivial property, we proceed as follows. Since, it is a nontrivial property, there will be a Turing machine that accepts that language. Given \( \Sigma \) and string \( \omega \), we then build a modified Turing machine that first simulates \( \Sigma \) on \( \omega \), and if it halts, then it takes an input string decides its acceptance. Hence, this modified Turing machine can decide the acceptance of the input string for the decision of its nontrivial property if and only if the halting problem can be solved. Hence, the nontrivial property is undecidable. This reduction of the halting problem to the nontrivial property problem is shown in Figure 4.11. See [HMU07] for more details of the proof.
4.2. DECIDABILITY AND COMPUTABILITY

Theorem 4.40 (String Accepted by Linear Bounded Automata). Given a string and a linear bounded automata, it is decidable whether that string will be accepted. Equivalently, the membership problem for a context sensitive grammar is decidable.

Proof. The number of configurations of a linear bounded automata are finite, since it can only use a finite fixed amount of tape, and hence there are only finite locations of the read-write head, and various symbol placements on the tape. Hence, it can be easily found if a configuration is repeated when this automata is running. The repeat of a configuration indicates an infinite loop, i.e. no halting.

Some other problems can also be shown to be undecidable such as deciding if a context sensitive language is empty, or the membership for a context free language. Another interesting related problem is the Post correspondence problem. Please refer to [Sip06] for details on these.

4.2.5 Many-one Reduction

By relating a language to another language via a computable function can help answer some decidability questions. For this formally, we use the concept of many-one reducibility.

Definition 4.41 (Many-one Reduction). Let a language $A$ be defined over the alphabet $\Sigma$ and language $B$ over the alphabet $\Gamma$. We say that language $A$ is many-one reducible (or m-reducible) to $B$ and show it as $A \leq_m B$ if $A = f^{-1}(B)$ for a (total) computable function $f : \Sigma^* \rightarrow \Gamma^*$. If the function $f$ is injective, we use the term 1-reducible and show it as $A \leq_1 B$.

Theorem 4.42 (m-reduction Decidability). If $A \leq_m B$, then $B$ is decidable $\Rightarrow A$ is decidable.
Proof. Given $\omega$, compute $f(\omega)$ and ascertain the membership for $f(\omega)$ in $B$, and that gives the answer for the same for $A$. □

**Corollary 4.43 (m-reduction Undecidability).** If $A \leq_m B$, then $A$ is undecidable $\Rightarrow$ $B$ is undecidable.

**Theorem 4.44 (m-reduction Semi-decidability).** If $A \leq_m B$, then $B$ is semi-decidable $\Rightarrow$ $A$ is semi-decidable. □

Proof. Given $\omega$, compute $f(\omega)$ and ascertain the membership for $f(\omega)$ in $B$. Then the semi-decidability of $B$ will create the same for $A$. □

**Corollary 4.45 (m-reduction Turing Semi-undecidability).** If $A \leq_m B$, then $A$ is semi-undecidable $\Rightarrow$ $B$ is semi-undecidable.

**Definition 4.46 (Turing Reduction).** Language $A$ is Turing reducible to $B$ shown as $A \leq_T B$ if the characteristic function of $A$ can be solved given an Oracle for the membership function for $B$. $A$ is called $B$-recursive, and if the function is a partial function then it is called $B$-recursively enumerable. □

### 4.3 Complexity

In this section we study the complexity of strings as well as of algorithms. For algorithms, we will look at at time and space complexity and also review intractability of problems.

#### 4.3.1 Kolmogorov Complexity

In this section we will study Kolmogorov complexity of a string and its various properties.

**Definition 4.47 (Kolmogorov Complexity).** Kolmogorov complexity of a binary string $\omega$ is the length of the minimum length description of a Turing machine $\mathcal{T}$ and an input string $\alpha$ such that when $\mathcal{T}$ operates on string $\alpha$ as an input string, $\mathcal{T}$ halts with $\omega$ as the output string. We use $K(\omega)$ to show the Kolmogorov complexity of $\omega$. □

Some fundamental results for Kolmogorov complexity is presented in theorems here. See [Sip06] for proofs and details.

**Theorem 4.48.** $\exists K \forall x, R(\omega) \leq \ell(\omega) + K$ □
4.4. Time Complexity

Here, we used $\ell(\omega)$ to express the length of the string $\omega$ as the number of symbols from its alphabet, $K$ a constant.

**Theorem 4.49.** $\exists \forall x, \hat{R}(\omega x) \leq \hat{R}(\omega) + K$

**Theorem 4.50.** $\exists \forall x, y, \hat{R}(\omega) + \hat{R}(\alpha) + K < \hat{R}(\omega \alpha) \leq 2\hat{R}(\omega) + \hat{R}(\alpha) + K$

### 4.3.2 Compressibility of Strings

We will consider binary strings, i.e. strings on alphabet $\{0, 1\}$ for studying compressibility.

**Definition 4.51** (string compressibility). If a string $s$ has the property

$$\hat{R}(s) \leq \ell(s) - \alpha$$

then it is called $\alpha$-compressible, otherwise, it is called incompressible by $\alpha$. A string that is incompressible by 1 is called incompressible.

**Theorem 4.52** (Incompressible String).

$$\forall n \in \mathbb{N}, \exists a \text{ string } s \text{ with } \ell(s) = n, \hat{R}(s) \geq \ell(s)$$

*In other words, for every $n$, there exists a string of length $n$ that is incompressible.*

**Proof.** The proof comes from counting, as the number of strings of length $n$ is $2^n$, and the descriptions of length less than $n$ is $1 + 2 + 2^2 + \ldots + 2^{n-1}$ which is $2^n - 1$.

Incompressible strings are many. In fact they share many properties with randomly chosen strings. Any computable property that is true for almost all strings, then for any $\alpha > 0$, that property if false for only finitely many strings that are incompressible by $\alpha$. Interestingly, it is undecidable if a string is incompressible.

### 4.4 Time Complexity

We studied complexity in Chapter 2. Here we will study it in more details and also more formally. We will use the various notions of asymptotic algorithmic complexity that we introduced in that chapter. Algorithms are defined in terms of Turing machine, and hence complexity for algorithms is studied in terms of Turing machines.
Time complexity can be defined for Turing machines, or for computable functions, or for languages decidable by Turing machines, etc. As we have seen, these concepts are interrelated and hence complexity can be decided in terms of any of these related concepts.

**Definition 4.53** (Time Complexity of a Turing Machine). Time complexity of a Turing machine \( \mathfrak{T} \) that halts on every input, is a function \( f : \mathbb{N} \to \mathbb{N} \) that shows the maximum number of steps taken to halt and produce an output when the length of the input string is \( n \).

**Definition 4.54** (Order of the Time Complexity of a Turing Machine). The order of the time complexity of a Turing machine \( \mathfrak{T} \) that halts on every input, is the order of the function \( f : \mathbb{N} \to \mathbb{N} \) that shows the maximum number of steps taken to halt and produce an output when the length of the input string is \( n \), as used in the Definition 4.53.

We can also define time complexity for languages by the order of the Turing machine that decides that language.

**Definition 4.55** (Time Complexity Class of a Language). The time complexity class for a total function \( f : \mathbb{N} \to \mathbb{N} \) for languages is the set of all languages that are decide by a Turing machine of order \( O(f(n)) \). Formally,

\[
\text{TIME}(f(n)) = \{ \mathcal{L} | \mathcal{L} \text{ is a language decided by an order } O(n) \text{ Turing machine} \}\]

As an example of time complexity of languages, it can be shown that any language that can be decided in time of the order \( o(n \log n) \) is regular.

We can also define complexity class of a set of computable functions that accomplish their computation within certain number of steps as a function of the size \( n \) of the problem. Complexity class of Boolean functions, i.e. functions from strings on an alphabet (or natural numbers) to the set \( \{0, 1\} \) also give rise to languages, as the set of strings for which that given function takes a value of say 1.

We will use the time complexity for an algorithm (equivalently a Turing machine that halts on every input in its domain), computable function, or language etc., and it will be clear from the context which one we are dealing with.
4.4. TIME COMPLEXITY

4.4.1 Time Complexity of Different Types of Turing Machines

We will compare the time complexity of multitape and nondeterministic Turing machines with that of a single track Turing machine, which will be our standard for complexity analysis. Nondeterministic Turing machine becomes very important in classifying algorithms in conjunction with the single tape deterministic Turing machine. As far as tractability of an algorithm is concerned though, we will see that the number (finite) of tapes does not make a difference.

Comparing Multitape and Single Tape Turing Machines

Figure 4.12 shows how the contents of a multitape Turing machine are copied into the single tape of a single tape Turing machine. The symbols to represent the content and location of the cells where the heads are are shown with dots, and also the symbol # has been used to demarcate between the contents of the two tapes. The relationship between the complexity of the single tape Turing machine that simulates a multitape Turing machine is given by the theorem below.

**Theorem 4.56 (Multitape to Single Tape Simulation).** A multitape Turing machine of the order $f(n)$ can be simulated by a single tape Turing machine of the order $f^2(n)$, assuming that the order of $f^2(n)$ is greater than $O(n)$. ☐
Proof. To copy the initial contents of the multitape to a single tape takes $O(n)$ steps. For each step of the multi-tape to simulate in the single tape takes $f(n)$ moves in space (on the tape). Hence total steps are $O(n) + f^2(n)$, and hence the complexity is $f^2(n)$ assuming that the order of $f^2(n)$ is greater than $O(n)$. □

Comparing Nondeterministic and Deterministic Turing Machines

Running time for a deterministic Turing machine is the number of steps the algorithm takes. Running time for a nondeterministic Turing machine is the number of steps the algorithm takes in the longest branch of its computations. Figure 4.13 shows the running time of $f(n)$ for a deterministic and a nondeterministic Turing machine computations.

![Diagram of Nondeterministic to Deterministic Turing Machine](diagram.png)

**Figure 4.13: Nondeterministic to Deterministic Turing Machine**

**Theorem 4.57** (Nondeterministic to Deterministic Turing Machine). A nondeterministic Turing machine of the order $f(n)$ can be simulated by a single tape deterministic Turing machine of the order $2^{O(f(n))}$.

Proof. The nondeterministic Turing machine will have at most $k$ branches at every node. The maximum number of leaves of the tree then will be $k^{f(n)}$. □
4.4. TIME COMPLEXITY

Hence, we have \( k^f(n) \) branches, and each branch takes \( f(n) \) computations. Hence, the complexity is \( O(f(n))O(k^f(n)) \). Our computations are as follows:

\[
O(f(n))O(k^f(n)) = O(f(n)k^f(n)) = O(f(n)2^af(n)) = O(f(n)2^f(n)) = O(2^{\log f(n)}f(n)) = O(2^{\log f(n)+f(n)}) = 2^{O(f(n))}
\]

4.4.2 P and NP Complexity

In this section we will study polynomial deterministic algorithms and languages, as well as nondeterministic polynomial algorithms.

**Definition 4.58** (Class P). The collection of problems (Turing machines, languages, algorithms) of class P are defined as:

\[
P = \bigcup_i \text{TIME}(n^i)
\]

Polynomial complexity algorithms are realistically solvable on practical computers, and different types of Turing machines are all equivalent under this complexity. Many important algorithms come under this class, such as matrix multiplication, path finding in graphs, dynamic programming algorithm, membership for context free language, etc.

**Example 4.59** (Matrix Multiplication). Consider a multiplication of two \( n \times n \) matrices.

\[
\begin{bmatrix}
a_{11} & a_{12} \\
a_{21} & a_{22}
\end{bmatrix}
\begin{bmatrix}
b_{11} & b_{12} \\
b_{21} & b_{22}
\end{bmatrix} =
\begin{bmatrix}
a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\
a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22}
\end{bmatrix}
\]

(4.18)

We get \( n^2 \) entires and each one has sum of \( n \) terms which has one multiplication each. For instance in the example shown here we have 4 terms where the first term is \( a_{11}b_{11} + a_{12}b_{21} \). Hence matrix multiplication complexity is of the order \( O(n^3) \), and hence belongs to class P.

To understand the class NP, we will study the Boolean satisfiability problem next.
CHAPTER 4. COMPUTATION THEORY

Boolean Satisfiability

We will consider Boolean variables that take values in the set \{0, 1\}. We use three operations of or, the operator shown as + called disjunction, and, the operator shown as \cdot called conjunction, as multiplication of real numbers, defined in the Table 4.1 and Table 4.2 below, where we show each operator in two different ways, and finally the operator not, showing negation for which we use an overbar on a Boolean variable. The not operator’s truth table is shown in Table 4.3.

<table>
<thead>
<tr>
<th>(A)</th>
<th>(B)</th>
<th>(A + B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(A)</th>
<th>(B)</th>
<th>(A \cdot B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.1: Boolean or

Table 4.2: Boolean and

Boolean expressions are expressions such as \((A + B) \cdot C\) using Boolean variables and the three operators. They are also called Boolean functions, as they map the Boolean values of \(A, B,\) and \(C\), into \{0, 1\}. These are also called Boolean statements in logic theory. These expressions get evaluated to either 0 or 1 after we assign Boolean values to all the variables in the expression. Since there three variables in the expression there are \(2^3 = 8\) possible allocations of values to the variables. Table 4.4 shows the values the expression \((A + B) \cdot C\) for each of these allocations.
Table 4.3: Boolean not

<table>
<thead>
<tr>
<th>A</th>
<th>( \overline{A} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.4: \((A + B) \cdot C\)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>((A + B) \cdot C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Now, we present some definitions formally (See [Arn11]).

**Definition 4.60** (Boolean Constant). A Boolean constant is a member of the set \( \{0, 1\} \).

**Definition 4.61** (Boolean Variable). A Boolean variable is a variable with range \( \{0, 1\} \).

**Definition 4.62** (Boolean Function). A Boolean function (also called Boolean statement, or expressions) is a function with all of its variables (independent and dependent) being Boolean.

The truth tables for the three operators are symmetric which leads to these operations being commutative. They can also be shown to be associative and distributive. Hence, the expression \((A + B) \cdot C\) can be simplified as

\[
(A + B) \cdot C = AC + BC
\]
These laws, commutative, associative, and distributive are presented next, in which we use the symbol $\equiv$ to show equivalence of two Boolean expressions. Equivalence indicates that both expressions produce the same exact Boolean function.

**Theorem 4.63** (Commutative Law).

\[
A + B \equiv B + A \\
AB \equiv BA
\]

**Theorem 4.64** (Associative Law).

\[
(A + B) + C \equiv A + (B + C) \\
(AB)C \equiv A(BC)
\]

**Theorem 4.65** (Distributive Law).

\[
A(B + C) \equiv AB + AC \\
A + (BC) \equiv (A + B)(A + C)
\]

This expression is in a Disjunctive Normal Form (DNF), which means that it is a disjunction of conjuncts such as $AC$ and $BC$. Conjuncts are built from Boolean variables and their negations by conjunction. For instance, the following are all conjuncts built from $A$, $B$, and $C$: $A\overline{B}C$, $BC$, etc.

We can also build a form which is the dual of the DNF. For this we need the following theorem (De Morgan’s Law). Conversion to this new form can be done by using distributive laws as well as De Morgan’s Law.

**Theorem 4.66** (De Morgan’s Law).

\[
A + B = \overline{AB} \\
A \cdot B = \overline{A + B}
\]
The expression \((A + B) \cdot C\) is already in this form which is the conjunction of \((A + B)\) and \((C)\). This expression is in a Conjunctive Normal Form (CNF), which means that it is a conjunction of disjuncts such as \(A + B\) and \(C\). Disjuncts are built from Boolean variables and their negations by disjunction. For instance, the following are all disjuncts built from \(A\), \(B\), and \(C\): \((A + \overline{B} + C)\), \((B + C)\), etc. If we had the expression \(AB + C\). We could convert it into a CNF form by using the second distributive law to get \((A + C)(B + C)\). A literal is a Boolean variable or its negation, a clause is the disjunction of literals. Hence, all the disjuncts such as \((A + \overline{B} + C)\), \((B + C)\), etc. are clauses and each one is built out of literals, such as \(A\), \(\overline{B}\), etc.

The **Boolean satisfiability** asks the following question. We are given a Boolean function in the CNF form with a fixed number of clauses \(k\). Find an assignment of Boolean values to all the variables in the expression such that the function will be satisfied, i.e. will have a value of 1. Now, we can verify if some assignment satisfies the function or not. Given \(n\) Boolean variables, \(A_i\) the assignment is a function that assigns to each variable a Boolean constant. So, for instance one assignment for \((A + B) \cdot C\) would be \(a(A) = 1, a(B) = 0,\) and \(a(C) = 1\). Once this assignment is given, then we can evaluate the function \((A + B) \cdot C\) to verify its value. The assignment is called a certificate for which we can compute the function value to verify if its value is 1. This verification problem is of the order \(O(n^p)\) for some \(p\) and hence has polynomial complexity. However, to find an assignment that has a value 1, can lead us to search every row of the Table 4.5, which has \(2^n\) rows, which in this case is 8 and which grows exponentially. The idea of the verifier algorithm given a certificate (the assignment, such as the row \((1, 0, 1)\) for our example, can be viewed as a nondeterministic Turing machine deciding the problem, and hence these problems are called NP, standing for nondeterministic polynomial.

**Definition 4.67** (Class NP). The collection of problems (Turing machines, languages, algorithms) of class NP are the ones with polynomial time verifiers.

**Theorem 4.68.** Language NP and the class of languages that are decided by nondeterministic Turing machines of polynomial time complexity are equivalent. In other words

\[
NP = \bigcup_i NTIME(n^i)
\]

where we have used NTIME to refer to the complexity of verification.

What is known about NP problems in terms of deciding them deterministically is that we can use exponentially complex algorithms, i.e.
NP ⊂ EXPTIME = ∪ \text{TIME}(2^{n^i}) \quad (4.20)

\textbf{Definition 4.69} (Polynomial Time Computable Function). A function is called polynomial time computable if \( \exists \mathcal{S} = (Q, \Sigma, \Theta, \delta, q_0, \Box, F) \), which is a polynomial time Turing machine such that

\[ q_0 u \Rightarrow \exists q_f f(u), q_f \in F, u, f(u) \in \Theta^* \quad (4.21) \]

It is assumed that \( u \in \text{domain of the function } f \).

\textbf{Definition 4.70} (Polynomial Time Many-one Reduction). Let a language \( A \) be defined over the alphabet \( \Sigma \) and language \( B \) over the alphabet \( \Gamma \). We say that language \( A \) is many-one polynomial time reducible (or p-reducible) to \( B \) and show it as \( A \leq_p B \) if \( A = f^{-1}(B) \) for a (total) polynomial time computable function \( f : \Sigma^* \rightarrow \Gamma^* \).

\textbf{Theorem 4.71.} If \( A \leq_p B \), then \( B \in P \Rightarrow A \in P \).

\textbf{Definition 4.72} (NP-complete). A language is in NP-complete if

1. it is in NP, and
2. every NP is polynomially reducible to it.

\textbf{Definition 4.73} (NP-hard). A language is in NP-complete if
4.4. TIME COMPLEXITY

• every NP is polynomially reducible to it.

We present one of the most important theorems in complexity theory next, the Cook-Levin theorem, that states that the The Boolean satisfiability (SAT) problem is NP-complete. Original proof of the Cook-Levin theorem was provided in the references [Coo71] and [Lev73], and twenty one problems were shown to be NP-complete in [Kar72]. Garey and Johnson show about three hundred problems to be NP complete in [MJ79]. The theorem and the essence of the proof as described in [MJ79] is presented below.

**Theorem 4.74** (Cook-Levin theorem). *The Boolean satisfiability (SAT) problem is NP-complete.*

**Proof:** To prove that the SAT problem is NP-complete, we have to show two things:

1. the SAT problem is in NP, and
2. every NP problem can polynomially reduced to SAT.

For the first part, we have to show that there is a polynomially times verifier for the SAT problem. We have already shown that in the introduction to the SAT problem. What is left to prove is that every NP problem can be reduced in polynomial time (steps) into the SAT problem. The proof of that depends on showing that a Turing machine description of any NP problem into a Boolean satisfiability expression is accomplished polynomially.

Fix a given NP problem and its corresponding nondeterministic Turing machine that verifies it in polynomial time \( p(n) \), for size \( n \) problem. The Turing machine verifies using only polynomial space and time steps and consequently we will require a bounded number of variables. The nondeterministic Turing machine \( \mathcal{T} \) is the given septuple \((Q, \Sigma, \Theta, \delta, q_0, \Box, F)\). Various states of \( \mathcal{T} \) are given by \( Q = \{q_1, \ldots, q_r\} \), and tape symbols by \( \Theta = \{\theta_1, \ldots, \theta_s\} \).

The SAT expression is built by using a set of variables \( \mathcal{U} \), and the set \( \mathcal{C} \) of possible clauses. As shown in [MJ79], we define variables as shown in the figure 4.6.

To represent the function of the Turing machine by a Boolean expression in the CNF form, we use the six types of conjunction of clauses in Table 4.7, as detailed in [MJ79].

To get an understanding of how these provide the mapping in polynomial time, we see that the conjunction \( C_1 \) is obtained from the conjunction of the following two conjunctions of clauses, to indicate that the state must be in at least one of the given states of the Turing machine as shown in the first statement, and that it can not be in more than one, as shown in the next.
Table 4.6: Variables for Representation

<table>
<thead>
<tr>
<th>State</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S(i,j)$</td>
<td>At time step $i$ is in state $q_j$</td>
</tr>
<tr>
<td>$R_w(i,j)$</td>
<td>At $i$ the read-write head is on cell $j$</td>
</tr>
<tr>
<td>$T(i,j,k)$</td>
<td>At $i$ cell $j$ has symbol $\theta_k$</td>
</tr>
</tbody>
</table>

Table 4.7: Clauses Collection

<table>
<thead>
<tr>
<th>State</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_1$</td>
<td>At time step $i$ is in a unique state</td>
</tr>
<tr>
<td>$C_2$</td>
<td>At time $i$ the read-write head is on a unique cell</td>
</tr>
<tr>
<td>$C_3$</td>
<td>At time $i$ each cell has a unique symbol</td>
</tr>
<tr>
<td>$C_4$</td>
<td>At time 0 is in state $q_0$</td>
</tr>
<tr>
<td>$C_5$</td>
<td>Each transition follows the transition rules $\delta$</td>
</tr>
<tr>
<td>$C_6$</td>
<td>The final state is in $F$</td>
</tr>
</tbody>
</table>

$(S(i,0) + S(i,1) + \ldots + S(i,r))$, where, $0 \leq i \leq p(n)$

$(S(i,j) + S(i,j'))$ where, $0 \neq i \neq p(n)$, $0 \leq j < j' \leq r$

Analysis shows (please refer to [MJ79] for details) that the number of variables in $\Sigma$ is $O(p(n)^2)$, and the number of clauses is also $O(p(n)^2)$. It shows that the length of the Boolean expression is of $O(p(n)^4)$, and hence, the given NP problem has been polynomially reduced to the SAT problem. □

What we have shown is that any NP problem can be polynomially reduced to the SAT problem, i.e. $\mathsf{NP} \implies \mathsf{SAT}$. Hence to show any problem $Q$ is also NP complete, it suffices to show that $Q \in \mathsf{NP}$, and that $\mathsf{SAT} \implies Q$.

Many important problems with polynomial time algorithms (see [Kar72]) are:

- Minimum Spanning Tree
4.4. **TIME COMPLEXITY**

- Shortest Path
- Minimum Cut
- Arc Cover
- Bipartite Matching
- Linear Equations

These are very important problems and algorithms for studying networks, and will be discussed in more details in Chapter 5.

Many problems have been shown to NP-complete (see [Kar72] and [MJ79]). Some of these are:

- SAT
- 0–1 Integer Programming
- Clique
- Set Packing
- Node Cover
- Set Covering
- Feedback Node Set
- Feedback Arc Set
- Directed Hamilton Circuit
- Undirected Hamilton Circuit
- 3-SAT
- Chromatic Number
- Clique Cover
- Exact Cover
- Hitting Set
- Steiner Tree
- 3-Dimensional Matching
- Knapsack
- Job Sequencing
- Partition
- Max Cut

We will get into details of many of these problems in chapters on graphs and networks, where the solutions to these problems become relevant.

### 4.4.3 Space Complexity

Just as we have time complexity of algorithms (Turing machines) and languages related to the time resource bounds needed, we also have space complexity dealing with space requirements for algorithms and decidability properties of languages.
**Definition 4.75** (Space Complexity of a Turing machine). Space complexity of a Turing machine is a function \( f : \mathbb{N} \rightarrow \mathbb{N} \) that given the size of the input, produces the maximum number of cell positions in the input tape that the Turing machine uses for the processing of the input.

**Definition 4.76** (Space Complexity of a Nondeterministic Turing machine). Space complexity of a nondeterministic Turing machine that halts on every input on every branch, is a function \( f : \mathbb{N} \rightarrow \mathbb{N} \) that given the size of the input, produces the maximum number of cell positions in the input tape that any branch of the Turing machine uses for the processing of the input.

We use the notation \( \text{DSPACE}(f(n)) \) to represent the set that contains all languages that are decidable by a deterministic Turing machine in \( O(f(n)) \). Similarly, we use the notation \( \text{NSPACE}(f(n)) \) to represent the set that contains all languages that are decidable by a nondeterministic Turing machine in \( O(f(n)) \).

There is one essential way that the space complexity is different from time complexity. Space on the tape can be used again, but time spent can not be reclaimed. Now for a nondeterministic Turing machine, there are \( O(2^{f(n)}) \) branches. If we take one of the branches, now at each step, we need to use a stack to keep the configuration. In step 1, we use \( O(f(n)) \) space, in the next step, we use \( O(f(n))/2 \), etc. The total amount used becomes \( f(n) \). To do this for every one of \( f(n) \) possibilities gives the complexity of order \( O(f^2(n)) \). This is the statement of Savitch's theorem (see [Sav70], [Sip06]).

**Theorem 4.77** (Savitch's Theorem). For \( f : \mathbb{N} \rightarrow \mathbb{N} \), such that the order of complexity of \( f(n) \) is at least \( \log n \), then

\[
\text{NSPACE}(f(n)) \subset \text{DSPACE}(f^2(n))
\]

**Definition 4.78** (PSPACE). The collection of problems (Turing machines, languages, algorithms) of class PSPACE are defined as:

\[
PSPACE = \bigcup_i \text{SPACE}(n^i)
\]

**Definition 4.79** (NDSPACE). The collection of problems (Turing machines, languages, algorithms) of class NDSPACE are defined as:

\[
\text{NDSPACE} = \bigcup_i \text{NSPACE}(n^i)
\]
4.4. **TIME COMPLEXITY**

**Corollary 4.80** (to Savitch’s Theorem). \( \text{NDSPACE}=\text{PSPACE} \)

**Definition 4.81** (EXPTIME). The collection of problems (Turing machines, languages, algorithms) of class EXPTIME are defined as:

\[
\text{EXPTIME} = \bigcup_i \text{TIME}(2^{n^i})
\]

**PSpace Completeness**

Every time-step of a Turing machine can claim only one new cell. Hence, the time complexity of a Turing machine can not be more than the space complexity. Moreover, we can see that the SAT problem is in \( \text{SPACE}(n) \), i.e., it has linear space complexity. Hence, we get the following relationship for different complexity classes.

\[
P \subset \text{NP} \subset \text{PSpace} = \text{NPSPACE} \subset \text{EXPTIME}
\]

**Definition 4.82** (PSpace-complete). A language is in PSPACE-complete if

1. it is in PSPACE, and
2. every PSPACE is polynomially time reducible to it.

**Definition 4.83** (PSpace-hard). A language is in PSPACE-complete if

- every PSPACE is polynomially time reducible to it.

**Quantified Boolean Formula (QBF)**

For the SAT problem we looked at satisfiability of Boolean formula. For instance we looked at a formula such as \((A + B) \cdot (B + C)\) to find any instance which would make the formula true, or have a value of logic 1. This problem can also be viewed as solving the quantified Boolean formula with existential quantifier, as

\[
\exists x_1 \exists x_2 \exists x_3 (x_1 + x_2) \cdot (x_2 + x_3)
\]

A quantified Boolean formula, in general contains both, existential and universal quantifiers, such as

\[
\forall x_1 \exists x_2 \forall x_3 (x_1 + x_3) \cdot (\overline{x_2} + x_3)
\]

The QBF problem is to find if a given QBF formula is true or not. The language QBF is the collection of true quantified Boolean formula.
Theorem 4.84. QBF is PSPACE-complete.

There are two parts to proving this theorem. In the first part, we need to show that the problem/language is in PSPACE. This is easy to show, since given an expression it can evaluate the expression iteratively over the quantifiers of the variables to see if the statement is true. It just needs to store the value of each variable, and that takes only linear space. The second part of the proof is more involved and is similar to the proof for Savitch's theorem. We omit the proof of this theorem. More details can be obtained from [Sip06].

We can compare NP-complete problems to solving puzzles, since they are related to solving satisfiability. Similarly, we can compare PSPACE-complete problems to solving games, since the existential and universal quantifiers can provide a game like structure for these problems.

Classes L and NL

It takes $O(n)$ complexity just to read an input of size $n$. However, by considering a read only tape as separate as the read-write tape, and only looking at the cells of the read-write tape for space complexity, we can study sublinear space problems.

Definition 4.85 (Class L).

\[ L = SPACE(\log n) \]

Definition 4.86 (Class NL).

\[ NL = NSPACE(\log n) \]
from the nondeterminism, and then just counting the nodes and moving to
the next node at each step from the ones available at each step.
If a Turing machine with read-only tape has input of length \( n \), uses space
\( f(n) \), has \( b \) tape symbols, then its space complexity is

\[
O(nf(n)b^{f(n)}) = O(nf(n)2^{af(n)}) = O(nf(n)2^{f(n)}) = O(n2^{\log f(n)}2^{f(n)}) = O(n2^{\log f(n)+f(n)}) = O(n2^{f(n)}) = O(2^{\log n2^{f(n)}}) = O(2^{\log n+f(n)}) = 2^{O(f(n))} \text{ if } f(n) \geq \log n
\]

A **log space computable function** is one that can be computed by a Turing
machine with a read-only input tape, a write-only output tape and a read-
write working tape that uses \( \log n \) space to compute the function. This tape
reads the input string \( s \) on its input tape, uses the working tape for its compu-
tations and then halts after placing the output string \( f(s) \) on the output tape.
A Turing machine with three tapes like this is shown in Figure 4.14.

**Figure 4.14:** Read-only,Write-only,Read-write Tape Turing Machine

**Definition 4.87** (NL-complete). A language is in NL-complete if
1. it is in NL, and
2. every NL is log space reducible to it.
Theorem 4.88. If \( A \leq_L B \), then \( B \in L \Rightarrow A \in L \). □

In sublinear languages although the input length is \( n \), but that is not counted in space requirement, since only the working tape space is used. However, in this theorem, the space requirement for representing the output of the mapping \( f(s) \) could cause a problem even if that is linear. So, hence, the first part of the computation and the second part have to work in a synchronized fashion so that only one symbol of \( f(s) \) is shown on the output tape one at a time. This in essence makes the space requirement of the \( f(s) \) to be \( O(1) \). This sub-linear simulation is shown in Figure 4.15.

![Figure 4.15: Sublinear Simulation](image)

**Directed Path in a Digraph** A directed graph \( G_D \) is a doublet \( (\mathcal{N}, \mathcal{E}) \), where \( \mathcal{N} \) is a finite set of nodes, \( \{n_1, \ldots, n_p\} \), and \( \mathcal{E} \) is a finite set of edges consisting of ordered pairs \( (n_i, n_j) \) from the set of nodes. An example is shown in Figure 4.16. We can code a given directed graph into a string (e.g. using binary symbols). A directed path from an origin node \( O \) to a destination node \( D \), where \( O \in \mathcal{N} \) and \( D \in \mathcal{N} \) is a string of nodes \( n_i, \ldots, n_r \), such that each pair of symbols in this string is in \( \mathcal{E} \). For example, Figure 4.16 has a path from node \( d \) to node \( g \).

The decision problem for this directed graph is the following. Given a directed graph \( G_D = (\mathcal{N}, \mathcal{E}) \), and two nodes called the origin node \( O \in \mathcal{N} \), and the destination node \( D \in \mathcal{N} \), does a directed path between these two nodes exist. This can be converted into a language as follows:

**PATH Membership Problem:** Given a string \( \omega \) that is a binary code for \( (G_D, O, D) \), does \( \omega \in \text{PATH} \), where, PATH is defined as:

\[
\text{PATH} = \{s | \text{s representing (}G_D, O, D)\text{, } \exists \text{ a directed path from } O \text{ to } D\}
\]

Theorem 4.89. \( \text{PATH} \in P \) □
4.4. TIME COMPLEXITY

![Directed Graph Path problem](image)

Figure 4.16: Directed Graph Path problem

Proof. We present the breadth first search based polynomial algorithm to accomplish this task.

1. Mark node $O$.
2. If any node emanating from a marked node is unmarked, then mark it.
   Perform this step iteratively, till no more nodes can be marked.
3. If $D$ is marked, a directed path exists, otherwise it does not exits.

The number of steps used in the algorithm are in the order $O(n)$ and hence polynomial in time.

The steps of the breadth first search are shown in the example in Figure 4.17. Figure 4.17a shows the origin node with double line, and the destination node with the dashed line. Each next step keeps marking, and in this example the destination node ends up getting marked as well. However, if the origin node was $a$, then the destination node of $g$ would not get marked.

We can see that the breadth first search based marking algorithm uses linear space, and there is a nondeterministic log space algorithm for it.

Theorem 4.90. $PATH$ is NL-complete.

We have already seen that PATH is in NL. What has to be shown is that every language in NL can be log space reduced to PATH. Every configuration of the nondeterministic Turing machine for the NL language can be represented by a node in the graph. For a string $\omega$ in the language we can construct a description of a graph and an origin and destination pair, such that the nondeterministic Turing machine for the NL language accepts the string if and only if there is a path from the origin to destination. Counting all the nodes and edges of the graph described by the string $\omega$ have complexity $O(\log n)$. We can also see that $NL \subset P$, because any Turing machine running in $\log n$ space runs
in $n^{2^{O(\log n)}}$ time, which is polynomial. It can also be shown that $\text{NL} = \text{coNL}$, where $\text{coNL}$ is the collection of languages whose complements are in $\text{NL}$.

### 4.5 Bibliographical Notes

We discuss bibliography related to this section ···

### 4.6 Exercises

**Problem 4.1** (incompressible by $\alpha$). There are at least $2^n - 2^{n-\alpha+1} + 1$ strings of length $n$ that are incompressible by $\alpha$. 

---

**Figure 4.17:** Breadth First Algorithm for PATH

(a) O-D Pair

(b) First Step Markings

(c) Second Step Markings

(d) Destination Marked
This chapter reviews the fundamentals elements of the theory, data structures and implementation of graphs and trees as well as the foundations of algebraic graph theory which is very important for studying resilient networks. The fundamental notions and theory of graphs is presented in many texts such as [Har] [Die05], [BM76], [Bol82], [CLZ10], [GY04], and [Bal97].

5.1 Graphs

Definition 5.1 (Graph). Graph is a pair \( G = (V, E) \), such that \( V \cap E = \emptyset \). The elements of the vertex set \( V \) are vertices (or nodes or points). The edge set \( E \) contains edges (or lines). The members of \( E \) are unordered pairs \( \{u, v\} \), where \( u, v \in V \). The vertex set of a graph \( G \) is shown as \( V(G) \) and the edge set of the same graph as \( E(G) \).

\[ \Box \]

Example 5.2 (Example Graphs). Some example graphs are shown in Figure 5.1. The sets \( V \) and \( E \) for these examples are:

1. In Figure 5.1a \( V = \{a, b, c\} \), and \( E = \{\{a, b\}, \{b, c\}, \{c, a\}\} \).
2. In Figure 5.1b \( V = \{a, b, c, d\} \), and \( E = \{\{a, b\}, \{b, c\}, \{c, d\}, \{d, a\}\} \).
3. In Figure 5.1c \( V = \{a, b, c, d\} \), and \( E = \{\{a, b\}, \{a, c\}, \{b, c\}, \{b, d\}, \{c, d\}, \{d, a\}\} \).

\[ \triangle \]
CHAPTER 5. GRAPHS AND TREES

5.1.1 Types of Graphs

Definition 5.3 (Digraph). Digraph, or a directed graph, is a pair $D = (V, E)$, such that $V \cap E = \emptyset$. The elements of $V$ are vertices (or nodes or points). The set $E$ contains directed edges (or arcs or directed lines) with arrows. The members of $E$ are ordered pairs $(u, v)$, where $u, v \in V$. □

Example 5.4 (Example Digraphs). Some example graphs are shown in Figure 5.2. The sets $V$ and $E$ for these examples are:

1. In Figure 5.2a $V = \{a, b, c\}$, and $E = \{(a, b), (b, c), (c, a)\}$.
2. In Figure 5.2b $V = \{a, b, c, d\}$, and $E = \{(a, b), (b, c), (c, d), (d, a)\}$.
3. In Figure 5.2c $V = \{a, b, c, d\}$, and $E = \{(a, b), (a, c), (b, c), (b, d), (c, d), (d, a)\}$.

Definition 5.5. An edge $\{a, b\}$ joins or connects vertices $a$ and $b$, and is said to be incident on the vertices $a$ and $b$. The vertices $a$ and $b$ are adjacent or neighbors and are incident on the edge $\{a, b\}$. An isolated vertex is incident to no edge. □
Definition 5.6. The order of a graph (or digraph) $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, shown as $|\mathcal{G}|$ is equal to $\text{card}(\mathcal{V})$. The size of a graph is indicated by $\|\mathcal{G}\|$ and is equal to $\text{card}(\mathcal{E})$. Graph $\mathcal{G}$ with $|\mathcal{G}| = m$ and $\|\mathcal{G}\| = n$ as $\mathcal{G}(m, n)$. Graphs are finite or infinite depending on the order or size. We will consider only finite graphs unless we explicitly state otherwise.

Definition 5.7. A null graph is either an order 0 graph, i.e. $\emptyset = (\emptyset, \emptyset)$ or it is an edgeless graph, also called empty graph. Any graph of order less than two is called trivial.

Definition 5.8 (Simple Graph). A simple graph is a graph with no loops, i.e. no edges adjacent to the same vertex. Figure 5.3 shows an example of a simple and a non-simple graph, Figure 5.3a being the simple one, and Figure 5.3b being the non-simple one.

![Figure 5.3: Simple and Non-Simple Graph](image)

Definition 5.9 (Multigraph). A multigraph is a triplet $\mathcal{G}_m = (\mathcal{V}, \mathcal{E}, \mathcal{I})$, such that $\mathcal{V} \cap \mathcal{E} = \emptyset$. The elements of the vertex set $\mathcal{V}$ are vertices and the edge set $\mathcal{E}$ contains edges as in the case of graphs. The mapping $\mathcal{I} : \mathcal{E} \rightarrow \mathcal{V} \times \mathcal{V}$, assigns to each edge its pair of non-identical incident vertices. Multigraphs can not have loops and but can have multiple edges between the same pair of vertices, because each vertex exists independently and the mapping $\mathcal{I}$ assigns its adjacent vertices as compared to each edge existing as a pair of its adjacent vertices as in graphs, which doesn't allow the existence of multiple edges between the same pair of vertices.

Definition 5.10 (Pseudograph). A pseudograph is a multigraph except that it has at least one loop.

We can similarly define simple digraph, multidigraph and pseudodigraph. Examples to compare a simple graph with a multigraph with a multigraph and a pseudograph are shown in Figure 5.4.
Definition 5.11 (Hypergraph). A hypergraph is a pair $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, such that $\mathcal{V} \cap \mathcal{E} = \emptyset$. $\mathcal{V}$ is the vertex set and $\mathcal{E}$ is the edge set. The members of $\mathcal{E}$ are subsets of $\mathcal{V}$, i.e. $\mathcal{E} \subseteq 2^\mathcal{V}$. Figure 5.5 shows an example of a hypergraph where the edges are subsets of the set of vertices of the graph. Two different types of visualizations are shown for the same hypergraph. Figure 5.5a shows the edges as subsets directly, and Figure 5.5b shows the edges connecting to multiple vertices.

Figure 5.5: Hypergraph

Definition 5.12 (Labeled Graph). A labeled graph is a graph for which each vertex is assigned a specific label. Figure 5.6 shows a simple graph with order 3 and size 2. There are three distinct graphs with these properties but only one unlabelled one.

5.1.2 Some Graph Properties

Definition 5.13 (Neighborhood of a Vertex). Following the notation in [HHM08], the set $N(v)$ of all vertices adjacent to a vertex $v$ is the open neighborhood of
a vertex. Set union of the open neighborhood of a vertex with the vertex is the closed neighborhood \( N[v] \) of that vertex. Figure 5.7 shows the open and closed neighborhoods of the vertex \( a \) in the two graphs. The open neighborhood of vertex \( a \) is \( \{b, d, e\} \), and the closed neighborhood is \( \{a, b, d, e\} \). Open neighborhood \( N(\Omega) \) of a set of vertices \( \Omega \) is the union of the neighborhoods of all the members of that set. Closed neighborhood \( N[\Omega] \) of \( \Omega \) is \( \Omega \cup N(\Omega) \). Figure 5.8 shows the open and closed neighborhoods of the set \( \{a, c\} \). □

**Definition 5.14** (Boundary of a Vertex Set). Boundary \( \partial V \) of a vertex set \( V \) is given by \( \partial V = N(V) − V \), where we are using the set difference for \( N(V) − V \), which means a set obtained from removing all members of \( V \) from \( N(V) \). In Figure 5.9, we can see that \( \partial\{a, e\} = \{b, c, d\} \). □

**Definition 5.15** (Oriented Graph). A simple graph whose edges are given directions is called an oriented graph. Another way to look at oriented graph
is as a digraph that has no symmetric pair of directed arcs, i.e. there is no ordered pair \((a, b) \in \mathcal{E}\) such that \((b, a) \in \mathcal{E}\). Figure 5.10 shows examples of digraphs and oriented graphs of order 3 and size 3.

\[\square\]

**Definition 5.16** (Graph Degrees). 1. **Vertex Degree**: Degree of a vertex \(\text{deg}(v)\) is given by:

\[
\text{deg}(v) = \text{card}(N(v))
\]

i.e. it is the size of the open neighborhood of the vertex.
2. **Maximum degree** of a graph $\Delta(\mathcal{G})$ is defined to be:

$$\Delta(\mathcal{G}) = \max\{\deg(v) | v \in V(\mathcal{G})\}$$

3. **Minimum degree** of a graph $\delta(\mathcal{G})$ is defined to be:

$$\delta(\mathcal{G}) = \min\{\deg(v) | v \in V(\mathcal{G})\}$$

Degree sequence of a graph $\mathcal{G}(\mathcal{S})$ of order $k$ is a $k$-sequence of vertex degrees in a descending order. In the graph of Figure 5.11, we have:

![Figure 5.11: Graph Degrees](image)

Table 5.1: Graph Degrees

<table>
<thead>
<tr>
<th>Graph</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{G}(\mathcal{S})$</td>
<td>{a, b, c, d, e}</td>
</tr>
<tr>
<td>$\mathcal{G}(\mathcal{S})$</td>
<td>4, 3, 3, 2, 2</td>
</tr>
<tr>
<td>$\Delta(\mathcal{G})$</td>
<td>4</td>
</tr>
<tr>
<td>$\delta(\mathcal{G})$</td>
<td>2</td>
</tr>
</tbody>
</table>
Theorem 5.17 (First Theorem of Graph Theory). The number of vertices with odd degree is even.

Proof. Every edge, such as this one ——, adds two to sum of the degrees of all nodes. Hence, the sum of degrees of all nodes is even. There are two types of nodes, one type with even degree, and the other with odd. The sum of degrees of all nodes with even degrees is even, and when we subtract the sum of degrees of all nodes with even degrees from the sum of degrees of all nodes, we get an even number which is equal to the sum of degrees of all nodes whose degrees are even.

5.1.3 Set Operations on Graphs

Definition 5.18 (Union of Graphs). A sequence of vertices of a graph is a walk.

5.1.4 Ambulation on Graphs

Definition 5.19 (Walk). A sequence of vertices of a graph is a walk.

Definition 5.20 (Path). A walk is a path if no vertex is repeated.

Definition 5.21 (Trail). A walk is a path if no edge is repeated.

Definition 5.22 (Cycle). A closed path is a cycle. By closed path we mean a path whose first and last vertices are the same and no other repeats of vertices are allowed.

Definition 5.23 (Circuit). A closed trail is a circuit. By closed trail we mean a trail which has the same starting and terminal vertices.

5.2 Bibliographical Notes

We discuss bibliography related to this section ···

5.3 Exercises

Problem 5.1. Give an example
CHAPTER 6

MATHEMATICAL PROGRAMMING AND OPTIMIZATION

This chapter reviews · · ·

6.1 Introduction

6.2 Bibliographical Notes

We discuss bibliography related to this section · · ·

6.3 Exercises

Problem 6.1. Give an example
This chapter reviews · · ·

7.1 Introduction

7.2 Bibliographical Notes
We discuss bibliography related to this section · · ·

7.3 Exercises

Problem 7.1. Give an example
Chapter 8

Network Calculus

This chapter reviews...

8.1 Introduction

8.2 Bibliographical Notes

We discuss bibliography related to this section...

8.3 Exercises

Problem 8.1. Give an example
This chapter reviews ···

9.1 Introduction

9.2 Bibliographical Notes

We discuss bibliography related to this section ···

9.3 Exercises

Problem 9.1. Give an example
Chapter 10

Network Dynamics and Control

This chapter reviews...

10.1 Introduction

10.2 Bibliographical Notes

We discuss bibliography related to this section...

10.3 Exercises

Problem 10.1. Give an example
This chapter reviews complex networks.

11.1 Introduction

11.2 Bibliographical Notes

We discuss bibliography related to this section.

11.3 Exercises

Problem 11.1. Give an example
This chapter presents a survey of the mathematical methods used for modeling and solutions for the traffic assignment problem. It covers the static (steady state) traffic assignment techniques as well as dynamic traffic assignment in lumped parameter and distributed parameter settings. Moreover, it also surveys simulation based solutions. The chapter shows the models for static assignment, variational inequality method, projection dynamics for dynamic travel routing, discrete time and continuous time dynamic traffic assignment and macroscopic Dynamic Traffic Assignment (DTA). The chapter then presents the macroscopic DTA in terms of the Wardrop principle and derives a partial differential equation for experienced travel time function that can be integrated with the macroscopic DTA framework.

12.1 Introduction

Traffic assignment is an integral part of the four-stage transportation planning process (see [Gaz74] and [PO72]) that includes:

1. **Trip Generation**: Trip generation models estimate the number of trips generated at origin nodes and/or the number of trips attracted to destination nodes based on factors such as household income, demograph-
CHAPTER 12. DYNAMIC TRAFFIC ASSIGNMENT: A SURVEY OF MATHEMATICAL MODELS AND TECHNIQUES

ics, and land-use pattern. This data is obtained using surveys conducted periodically.

2. **Trip Distribution**: From the total number of trips generated and attracted at each node, trip distribution algorithms generate an origin-destination (O-D) matrix, in which each cell entry indicates the number of trips from one specific origin to one specific destination. Hitchcock model [Hit41], opportunity model [Sto40], gravity model [Voo56], and entropy models [Wil67] have been used for trip distribution algorithms.

3. **Modal Split**: Modal split analysis takes each cell value in the O-D matrix and divides it among various alternate modes of travel. The models are built based on performing discrete choice analysis on survey data (see [BAL85]).

4. **Traffic Assignment**: This step assigns each O-D flow value onto various alternate paths from that specific origin to the destination node. Assignments is based on optimization, usually using either Wardrop’s user-equilibrium ([War52], [She85]) or system optimum.

This four-step process comes from the traditional transportation planning area and is not designed for real-time operations, such as traffic responsive real-time incident management. However, a lot of research has taken place in the area of traffic assignment, especially dynamic traffic assignment that enables researchers to study transient traffic behavior, not just steady state one which the static assignment is designed for. A survey paper [PZ01] provides an excellent survey for the research work that has been performed in the area of dynamic traffic assignment. This chapter, in contrast to that survey work provides a survey of the mathematical framework that has been used in this area, and presents the results to enable the reader to grasp the various mathematical tools that have been used to study and analyze this problem. The models and approaches that have been used are varied, and this review chapter brings them together in order for the readers to see them in a somewhat linear fashion.

Outline  The remainder of this chapter is organized as follows. Section 12.2 gives account of various mathematical programming based static traffic assignment models that have been used. This section presents the user-equilibrium and system optimal formulations of the assignment problems, followed by the numerical schemes that have been used to solve those problems. Section 12.3 presents the fundamentals of the variational inequality framework which subsumes the mathematical programming methodology. Dynamic extension of the variational inequality framework is presented in the section 12.4. Section 12.5 presents the dynamic traffic assignment in continuous time. The discrete time and continuous time versions of this are presented. Section 12.7 presents
12.2. MATHEMATICAL PROGRAMMING FOR STATIC TRAFFIC ASSIGNMENT

the macroscopic DTA model including the new formulation and a new travel
time partial differential equation. Section 12.8 presents a brief summary of
the main features of simulation based DTA. Finally, Section 12.10 gives the
conclusions.

12.2 Mathematical Programming for Static Traffic Assignment

To build the mathematical framework for our chapter, we will start with ter-
minality and framework used in [She85]. We illustrate a sample network that
is also taken from [She85] and is shown in Figure 12.1. The digraph shows
four nodes and four arcs. Nodes 1 and 2 are origin nodes and node 4 is the
destination node. Hence there are two O-D pairs: 1 – 4 and 2 – 4.

<table>
<thead>
<tr>
<th>Table 12.1: Network Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>R</td>
</tr>
<tr>
<td>S</td>
</tr>
<tr>
<td>K</td>
</tr>
<tr>
<td>xa</td>
</tr>
<tr>
<td>ta</td>
</tr>
<tr>
<td>frs</td>
</tr>
<tr>
<td>cks</td>
</tr>
<tr>
<td>qrs</td>
</tr>
<tr>
<td>δa,k</td>
</tr>
</tbody>
</table>
There are two main classical traffic assignment optimization problems considered. Those two are: user-equilibrium, and system optimum.

### 12.2.1 User-equilibrium

User-equilibrium problem is based on Wardrop’s principle [War52] which is stated as:

*The journey times on all the routes actually used are equal, and less than those which would be experienced by a single vehicle on any unused route.*

This equilibrium condition can be obtained as a solution of a mathematical programming problem presented below [She85].

**Mathematical Programming Formulation**

The user equilibrium problem is stated as the mathematical programming problem (see [She85], [DS69b]) shown in Equation 12.1.

\[
\min z(x) = \sum_a \int_0^{x_a} t_a(\omega) d\omega \quad (12.1)
\]

with the equality constraints:

\[
\sum_k f_k^{rs} = q_{rs} \quad \forall r, s \quad (12.2)
\]

\[
x_a = \sum_r \sum_s \sum_k f_k^{rs} \delta_{a,k}^{rs} \quad (12.3)
\]

and the inequality constraint

\[
f_k^{rs} \geq 0 \quad \forall r, s \quad (12.4)
\]

The formulation given in Equation 12.1 is the Beckmann transformation [BMW55]. The link performance function \( t_a(x_a) \) is a function of traffic flow.
12.2. MATHEMATICAL PROGRAMMING FOR STATIC TRAFFIC ASSIGNMENT

on the link and the link capacity $c_a$. According to the Bureau of Public Roads (BPR) it is given by Equation 12.5

$$t_a(x_a) = v_f \left( 1 + 0.15 \left( \frac{x_a}{c_a} \right)^4 \right)$$  \hspace{1cm} (12.5)

The plot of a BPR function is shown in Figure 12.2

![Figure 12.2: BPR Link Performance Function](image)

**Wellposedness** The objective function is a smooth convex function ($\nabla^2 (x)$ is positive definite), and the feasible region is convex, hence a unique solution exists.

**Equivalence with Wardrop User-equilibrium Condition**

The Kuhn-Tucker conditions for the mathematical programming problem given by Equation 12.1 can be obtained in terms of the Lagrangian given in Equation 12.6.

$$\mathcal{L}(f, \lambda) = z[x(f)] + \sum_{rs} \lambda_{rs} \left( q_{rs} - \sum_k f_k^{rs} \right)$$  \hspace{1cm} (12.6)

Here, $\lambda_{rs}$ is the Lagrangian multiplier. The Kuhn-Tucker conditions $\forall k, r, s$ are:
Applying these necessary conditions 12.7 to the mathematical program 12.1 we obtain the Wardrop conditions \( \forall k, r, s \) as:

\[
f_{k}^{rs}(c_{k}^{rs} - u_{rs}) = 0
\]

\[
c_{k}^{rs} - u_{rs} \geq 0
\]

\[
\sum_{k} f_{k}^{rs} = q_{rs}
\]

\[
\sum_{k} f_{k}^{rs} \geq 0
\]

### 12.2.2 System Optimal Solution

System optimal solution is a solution that provides the total minimum time for the entire network. This condition can be obtained as a solution of a mathematical programming problem presented below [She85].

**Mathematical Programming Formulation**

The system optimal problem is stated as the mathematical programming problem (see [She85], [DS69b]) shown in Equation 12.9.

\[
\min z(x) = \sum_{a} x_{a} f_{a}(x_{a})
\]

with the equality constraints:

\[
\sum_{k} f_{k}^{rs} = q_{rs} \forall r, s
\]

\[
x_{a} = \sum_{r} \sum_{s} \sum_{k} f_{k}^{rs} \delta_{a,k}
\]

and the inequality constraint

\[
f_{k}^{rs} \geq 0 \forall r, s
\]
12.2. MATHEMATICAL PROGRAMMING FOR STATIC TRAFFIC ASSIGNMENT

Wellposedness The objective function is a smooth convex function ($\nabla^2(x)$ is positive definite), and the feasible region is convex, hence a unique solution exists.

Equivalence with Marginal User-equilibrium Condition

Applying Kuhn-Tucker conditions in this case we get $\forall k, r, s$:

$$f_k^{rs}(\tilde{c}_k^{rs} - \tilde{u}_rs) = 0$$

$$\tilde{c}_k^{rs} - \tilde{u}_rs \geq 0$$

$$\sum_k f_k^{rs} = q_{rs}$$

$$\sum_k f_k^{rs} \geq 0$$

(12.13)

Here, we have

$$\tilde{c}_k^{rs} = \sum_a \delta_{a,k}^{rs} \tilde{t}_a$$

(12.14)

where

$$\tilde{t}_a(x_a) = t_a(x_a) + x_a \frac{dt_a(x_a)}{x_a}$$

(12.15)

12.2.3 Numerical Schemes

The numerical scheme for solving user-equilibrium is based on the Frank-Wolfe algorithm that obtains the feasible direction and the maximum step-size for each iteration in one step. In fact, for the static traffic assignment problem, this amounts to simply applying all or nothing assignment to the shortest path for each O-D pair. The next step for each iteration involves finding the step size in the direction of the link flow solution of the all-or-nothing assignment step. Appropriate stopping criterion can be applied using some convergence principle. Details of this are provided in section 5.2, pages 116 – 122 of [She85].

There are heuristic numerical methods available to perform the assignment to achieve user-equilibrium. Two of the common heuristic techniques are:

**FHWA (modified capacity restraint) method** In this method at each iteration an all-or-nothing assignment of the entire OD flow is performed on a
single path. Travel times are updated by performing a weighted average of the travel time obtained by the latest assignment and the previous one. A convergence criterion is used to stop the iteration steps (for instance when the maximum difference between two iterative steps of link flows is less than some $\epsilon$). The final link flows assigned to the network are obtained by averaging the values from the last four iterative steps.

**Incremental Assignment** In incremental assignment, the OD values are divided into $n$ parts, and then each part is assigned to the network using all or nothing assignment based on the previous travel time values.

Dafermos ([DS69a]) applied the Frank Wolfe method to traffic assignment problem. This method also results in an all or nothing assignment, followed by a line search step in each iteration. The details can be obtained from [She85].

### 12.3 Variational Inequality based Static Traffic Assignment Model

Variational inequality formulation for traffic equilibrium has been used as it generalizes the framework of mathematical programming even when the travel time function on one link depends on the conditions on other links as well ([Daf80]). Once the variational inequality model has been formulated, it can be solved using some appropriate numerical scheme, such as the ones based on projection method, linear approximation, relaxation method, or the more general iterative scheme of Dafermos ([Daf83]).

The variational inequality problem is stated as:

**VI Problem:** Given a continuous function $f : \mathcal{K} \to \mathbb{R}^n$, where $\mathcal{K}$ is a given closed and convex subset of $\mathbb{R}^n$, $\langle \cdot, \cdot \rangle$ denotes the inner product, find $x \in \mathcal{K}$, such that

\[ \langle f(x), y - x \rangle \geq 0, \forall y \in \mathcal{K} \quad (12.16) \]
12.3. VARIATIONAL INEQUALITY BASED STATIC TRAFFIC ASSIGNMENT MODEL

Figure 12.3: Variational Inequality

Figure 12.3 shows a convex set and the variational inequality condition at a corner. The relationship between variational inequalities and optimization problems is given by the following two theorems ([KS00]).

**Theorem 12.1.** \( x \in \mathbb{K} \) s.t. \( f(x) = \min_{y \in \mathbb{K}} f(y) \iff \langle \nabla f(x), y - x \rangle \geq 0, \forall y \in \mathbb{K}. \)

**Theorem 12.2.** Convex \( f \) s.t. \( \langle \nabla f(x), y - x \rangle \geq 0, \forall y \in \mathbb{K} \implies f(x) = \min_{y \in \mathbb{K}} f(y). \)

To understand the constrained optimization problem and its interplay with variational inequalities, we present two figures (Figure 12.4 and Figure 12.5). The first quadrant in the \( x - y \) plane is the constrained region of search where we have assumed that \( h(x,y) \geq 0 \) is satisfied. The function to be minimized is given by \( f(x,y) \). Figure 12.4 shows the case when the minimizing point (on the \( x - y \) plane) for a given smooth cost function \( f(x,y) \) is contained in the interior of the region \( \mathbb{K} \) given by \( h(x,y) \geq 0 \). For the local minimum to exist, it is necessary that the gradient of the function is zero. Figure 12.5 shows the case when the minimizing point (on the \( x - y \) plane) for a given smooth cost function \( f(x,y) \) is contained at the boundary of the region \( \mathbb{K} \) given by \( h(x,y) = 0 \). For the given point to be the minimizer, any movement from this point in any feasible direction, i.e. in the direction of increasing \( h(x,y) \), should increase the value of \( f(x,y) \). This is the variational inequality statement. Moreover, in this case (when certain regularity conditions are satisfied ([Avr03])), since, the boundary is given by \( h(x,y) = 0 \), the directional derivative of \( f(x,y) \) in the direction of the tangent to the boundary should be zero. Moreover, the gradient of \( h(x,y) \) as well as that of \( f(x,y) \) should be pointing in the same direction. Kuhn-Tucker conditions (and Lagrangian method) state the condition on the relationship between the gradient of the cost function and that of the constraint functions. However, those are necessary conditions only if the problem satisfies certain regularity conditions (see [Avr03], [BSS06], and [Man94]).
CHAPTER 12. DYNAMIC TRAFFIC ASSIGNMENT: A SURVEY OF MATHEMATICAL MODELS AND TECHNIQUES

Figure 12.4: Minimizer in the Interior

Figure 12.5: Minimizer on the Boundary
12.3. VARIATIONAL INEQUALITY BASED STATIC TRAFFIC ASSIGNMENT MODEL

The theorems 12.1 and 12.2 demonstrate that variational inequality framework is more general than the mathematical programming framework. The variational inequality formulations of the traffic equilibrium (user) problems are stated below.

**Theorem 12.3.** \( x \in K \) is a solution to the user equilibrium problem if and only if

\[
\sum_{w \in W} \sum_{p \in P_w} C_p(x)(y - x) \geq 0, \forall x \in K
\]

Here, \( C_p \) is the travel time for the path \( p \) from the OD pair \( P_w \) from the set of OD pairs \( W \). This variational inequality can also be written in terms of traffic flows instead of link flows ([Nug00]).

To understand how the variational inequality formulation is more general than the optimization problem, consider the variational inequality again.

\[
\langle f(x), y - x \rangle \geq 0, \forall y \in K \tag{12.17}
\]

Now, if \( f(x) = \nabla \theta(x) \), then the condition

\[
\langle \nabla \theta(x), y - x \rangle \geq 0, \forall y \in K \tag{12.18}
\]

is the necessary condition for the optimization problem

\[
\text{minimize } \theta(x), \ x \in K \tag{12.19}
\]

The variational inequality has a corresponding gradient relationship based on the following theorem that is about the symmetry of second partial derivatives ([FP03]).

**Theorem 12.4.** Given \( f : K \rightarrow \mathbb{R}^n \), a continuously differentiable function on the open convex set \( K \subseteq \mathbb{R}^n \), then the following three conditions are equivalent.

1. \( \exists \theta, \text{ s.t. } f(x) = \nabla \theta(x) \)
2. \( \nabla f(x) = [\nabla f(x)]^T \ \forall x \in K \)
3. \( f \) is integrable on \( K \)

Theorem 12.4 shows that if the function \( f \) has a symmetric Jacobian then there is a corresponding optimization problem associated with it. However, if the Jacobian is asymmetric, for instance, when the user-equilibrium cost
is asymmetric with respect to traffic flows, then the Wardrop solution (variational inequality) is the framework without a corresponding mathematical programming problem.

On a cautionary note, Kuhn-Tucker conditions (and Lagrangian method) state the condition on the relationship between the gradient of the cost function and that of the constraint functions. However, those are necessary conditions only if the problem satisfies certain regularity conditions (see [Avr03], [BSS06], and [Man94]). For instance Figure 12.6 shows a function $f(x, y) = -x$ to be minimized which at the minimum point $(x, y) = (1, 0)$ does not satisfy the Kuhn-Tucker conditions for the region constrained by the first quadrant and the curve $y = 1 - x^3$.

![Figure 12.6: Violation of Kuhn-Tucker Condition](image)

### 12.4 Projected Dynamical Systems: Dynamic Variational Equation Model

Dynamics of route switching has been analyzed using dynamic variational inequality by Nagurney and Zhang ([NZ96], [ZN95], [NZ97], [ZN96], and [Daf88]). They developed the theory for projected dynamical systems in [ZN95], and applied the theory to traffic assignment in [ZN96] and [NZ97]. The paper by Dupuis and Nagurney ([DN93]) shows the main results in the theory and
applications of projected dynamical systems including its relationship to the Skorokhod problem ([Sko61]) for the study of its wellposedness.

Since variational inequality is related to the solution of a fixed point problem, we can related the variational inequality solution to be the equilibrium point of a dynamic system. The stability of the equilibrium point can be studied within the framework of this dynamic system, and then those dynamics can be used to model a time varying route assignment problem. This is precisely what Nagurney and Zhang do in their various papers. We summarize the technical results here.

### 12.4.1 Dynamic Route Choice

The dynamics of route choice adjustment are given by ([NZ96]):

\[
\dot{x} = \Pi_K(x, -C(x))
\]

(12.20)

where

\[
\Pi_K(x, v) = \lim_{\epsilon \to 0} \frac{P_K(x + \epsilon v) - x}{\epsilon}
\]

(12.21)

and

\[
P_K(x) = \text{Arg min}_{z \in \mathbb{K}} \|x - z\|
\]

(12.22)

Figure 12.7 shows the convex region inside which the vector field of the dynamics are shown. The equilibrium point as well as the solution of the variational inequality is at (0,0).

\[\begin{array}{c}
\textbf{Figure 12.7: The vector field}
\end{array}\]

The path flow vector \(x^* \in \mathbb{K}\) is the solution of

\[
0 = \Pi_K(x^*, -C(x^*))
\]

(12.23)
if and only if it satisfies
\[
\langle C(x^*), x - x^* \rangle \geq 0, \forall y \in \mathbb{K}
\] (12.24)

The following theorem from [NZ96] gives the condition for asymptotic stability of the equilibrium point of the projected dynamics related to the route adjustment process.

**Theorem 12.5.** *If the link cost is a strictly monotonic continuous function of link flows, then the equilibrium point for dynamics shown in Equation 12.20 is asymptotically stable.* □

The major result from [NZ96] for applying the discrete algorithm for the dynamic route choice problem is the following.

**Theorem 12.6.** *The Euler method given by*
\[
x^{\tau+1} = P_{\mathbb{K}}(x^\tau - a_\tau C(x^\tau))
\] (12.25)

*when*
\[
\lim_{\tau \to \infty} a_\tau = 0
\] (12.26)

*and*
\[
\sum_{\tau=1}^{\infty} a_\tau = \infty
\] (12.27)

*for \(\mathbb{K}\) being the positive orthant converges to some traffic network equilibrium path flow.* □

### 12.5 Dynamic Traffic Assignment

There are some nice reviews that provide summary of the models and work that has been performed in the area of Dynamic Traffic Assignment (DTA), such as [CBM+09], [PZ01], [RB96], and [Fri01]. Our review will focus on the mathematical aspects of these developments.

#### 12.5.1 Dynamic Traffic Assignment: Discrete Time

Merchant and Nemhauser ([MN78a] and [MN78b]) were the first to present a dynamic traffic assignment problem where time varying O-D flows are considered. Their formulation uses a state difference equation to represent the link dynamics, a conservation equation at the nodes of the digraph, and a cost function to minimize which leads to the following mathematical programming problem.
12.5. DYNAMIC TRAFFIC ASSIGNMENT

\[
\begin{align*}
\min z(x) &= \sum_{i=1}^{I} \sum_{j=1}^{a} t_{ij}(x_{ij}) \\
\text{with the link discrete time dynamics as equality constraints:} \\
x_{j}[i+1] &= x_{j}[i] - g_{j}(x_{j}[i]) + d_{j}[i], \ i = 0, 1, \cdots I - 1, \ \forall \ j \in \mathcal{A} \\
\text{the node conservation equation as} \\
\sum_{j \in \mathcal{A}(q)} d_{j}[i] &= F_{q}[i] + \sum_{j \in \mathcal{B}(q)} g_{j}(x_{j}[i]), \ i = 0, 1, \cdots I - 1, \ \forall \ q \in \mathcal{N} \\
\text{and the inequality constraints} \\
x_{j}[i] &\geq 0 \ i = 0, 1, \cdots I - 1, \ \forall \ j \in \mathcal{A} \\
d_{j}[i] &\geq 0 \ i = 0, 1, \cdots I - 1, \ \forall \ j \in \mathcal{A} \\
x_{j}[0] &= x_{0}[j] \ \forall \ j \in \mathcal{A}
\end{align*}
\]

Here, \(x_{j}[i]\) is the number of vehicles at the beginning of time period \(i\) in link \(j\), \(g_{j}(x_{j}[i])\) is the number of vehicles exiting the link in the unit time as a function of \(x_{j}[i]\), and \(d_{j}[i]\) is the number of vehicles entering the link \(j\). This problem formulation is a single destination network model. \(F_{q}[i]\) show the inflow rates as the time varying O-D flows. This can be extended to a multi origin multi destination formulation.

12.5.2 Dynamic Traffic Assignment: Continuous Time

Now we present a continuous time formulation of the DTA problem ([BLR01]) where a dynamic variational inequality is used. The traffic dynamics utilizes ordinary differential equations instead of finite difference equation as was the case for the discrete time formulation. There are other models that use dynamic continuous time models in optimal control or variational setting such as [FLT89], [FBS+93], and [Che99].

The time dependent Wardrop condition for the DTA are

\[
\begin{align*}
\int f_{k}^{rs}(t)(c_{k}^{rs}(t) - u_{rs}(t)) &= 0 \\
c_{k}^{rs}(t) - u_{rs}(t) &\geq 0 \\
\sum_{k} f_{k}^{rs}(t) &= q_{rs}(t) \\
\sum_{k} f_{k}^{rs}(t) &\geq 0
\end{align*}
\] (12.34)
The traffic dynamics for this DTA problem are the continuous version of the difference equation for the Merchant Nemhauser model, and are given by the following conservation ordinary differential equation.

\[
\dot{x}_{rks}(t) = u_{rks}(t) - g_{rks}(x_{ra}(t)) \tag{12.35}
\]

Here, \(u_{rks}(t)\) is the time varying inflow to link \(a\) on path \(k\) from origin \(r\) to destination \(s\), and \(g_{rks}(x_{ra}(t))\) is the corresponding time varying outflow which is the exit function which depends on the link density \(x_{ra}(t)\).

We have the following equality among matching constraints for various flows and links ([BLR01]).

\[
\sum_r \sum_s \sum_k x_{rks}(t) \delta_{rks} = x_{ra}(t) \tag{12.36}
\]

Numerical techniques are available to solve this variational inequality (see [BLR01]). Optimal control formulation for this problem can also be obtained which can be solved by calculus of variations or dynamic programming methods.

### 12.6 Travel Time and FIFO Issue

One major issue in dynamic traffic assignment problem is that of First In First Out (FIFO) constraint as discussed in [Car92]. According to FIFO if \(x_{t\tau a} > 0\) where \(x_{t\tau a}\) is the traffic flow that enters link \(a\) at time \(t\) and exits at time \(\tau\), then any flow that enters before time \(t\) can not exit after time \(\tau\) at an average. This condition is shown to be nonconvex in [Car92] and is presented in Equation 12.37.

\[
(x_{t\tau a} > 0) \Rightarrow \left( \sum_{t'\tau' a} x_{t'\tau' a} | t' < t, \tau' > \tau \right) = 0 \tag{12.37}
\]

A violation of this condition is shown in Figure 12.8. The violation essentially occurs because of the nature of the exit function and also the time and space discretization of the traffic link and dynamics. Both of these issues get resolved by a proper choice of space and time discretization that are chosen after the original modeling is performed in a hydrodynamic setting using the dynamic distributed parameter traffic flow theory. This theory allows for a proper development of a travel time function as well as a travel time vector field. This development is the main original technical contribution of this chapter.
12.7 Macroscopic Model for DTA

We propose to use a hydrodynamic traffic model in the framework of the DTA problem. The Lighthill-Whitham-Richards (LWR) model, named after the authors in [LW55] and [Ric56], is a macroscopic one-dimensional traffic model. The conservation law for traffic in one dimension is given by

$$\frac{\partial}{\partial t} \rho(t, x) + \frac{\partial}{\partial x} f(\rho(t, x)) = 0 \quad (12.38)$$

In this equation $\rho$ is the traffic density (vehicles or pedestrians) and $f$ is the flux which is the product of traffic density and the traffic speed $v$, i.e. $f = \rho v$. There are many models researchers have proposed for how the flux should be dependent on traffic conditions. This relationship is given by the fundamental diagram.

12.7.1 Greenshield’s Model

Greenshield’s model (see [Gre35]) uses a linear relationship between traffic density and traffic speed.

$$v(\rho) = v_f (1 - \frac{\rho}{\rho_m}) \quad (12.39)$$

where $v_f$ is the free flow speed and $\rho_m$ is the maximum density. Free flow speed is the speed of traffic when the density is zero. This is the maximum speed. The maximum density is the density at which there is a traffic jam and the speed is equal to zero. The flux function is concave as can be confirmed by noting the negative sign of the second derivative of flow with respect to density, i.e. $\frac{\partial^2 f}{\partial \rho^2} < 0$. The fundamental diagram refers to the relationship.
that the traffic density $\rho$, traffic speed $v$ and traffic flow $f$ have with each other. These relationships are shown in Figure 12.9.

![Figure 12.9: Fundamental Diagram using Greenshield Model](image)

### 12.7.2 Generalized/Weak Solution for the LWR Model

The hyperbolic Partial Differential Equation (PDE) for the LWR model given by Equation 12.38 can be solved by using the method of characteristics ([LeV94]). Figure 12.10 shows a $x - t$ plot for traffic density $\rho(t, x)$. Initially the traffic density is constant at $\rho_0$. At time $t = 0$, there is a traffic light at $x = 0$ that turns red. We see the shockwaves travelling backward so that there is a discontinuity between traffic density being $\rho_0$ to the left of the shock line and being $\rho_m$ to the right of it. On the right there is another shockwave travelling to the right between zero traffic density and $\rho_0$. At time $t = t_c$, the light turns green and we see rarefaction of traffic starting at $x = 0$. Corresponding to time $t = t_u$ we see the plot of traffic density $\rho(t_u, x)$ that shows to the two shock waves as well as rarefaction of the traffic density. This shows that the traffic solution has discontinuities and a weak solution of the LWR model is required that allows for these discontinuous solutions.
Generalized Solutions

For a conservation law

\[ \rho_t + f(\rho)_x = 0 \]  

(12.40)

with initial condition

\[ \rho(x,0) = \rho_0(x), \]  

(12.41)
where \( u_0(x) \in L^1_{loc}(\mathbb{R}; \mathbb{R}^n) \), solution in the distributional sense is defined below for smooth vector field \( f: \mathbb{R}^n \to \mathbb{R}^n \) (see [Bre05]).

**Definition 12.7.** A measurable locally integrable function \( \rho(t,x) \) is a solution in the distributional sense of the Cauchy problem 12.40 if for every \( \phi \in C_0^\infty(\mathbb{R}^+ \times \mathbb{R}) \)

\[
\int_{\mathbb{R}^+ \times \mathbb{R}} \left[ \rho(t,x) \phi_t(t,x) + f(\rho(t,x)) \phi_x(t,x) \right] \, dx \, dt + \int_{\mathbb{R}} u_0(x) \phi(x,0) \, dx = 0 \tag{12.42}
\]

**Weak Solutions**

A measurable locally integrable function \( u(t,x) \) is a weak solution in the distributional sense of the Cauchy problem (12.40) if it is a distributional solution in the open strip \((0,T) \times \mathbb{R}\), satisfies the initial condition 12.41 and if \( u \) is continuous as a function from \([0,T]\) into \( L^1_{loc} \). We require \( u(t,x) = u(t,x^+) \) and

\[
\lim_{t \to 0} \int_{\mathbb{R}} |u(t,x) - u_0(x)| \, dx = 0 \tag{12.43}
\]

**12.7.3 Scalar Initial-Boundary Problem**

Consider the scalar conservation law given here.

\[
u_t + f(t,x,u)x = 0 \tag{12.44}
\]

with initial condition

\[
u(0,x) = u_0(x), \tag{12.45}
\]

and boundary conditions

\[
u(t,a) = u_a(t) \text{ and } u(t,b) = u_b(t), \tag{12.46}
\]

The boundary conditions cannot be prescribed point-wise, since characteristics from inside the domain might be traveling outside at the boundary. If there are any data at the boundary for that time, that has to be discarded. Moreover, the data also must satisfy entropy condition at the boundary so as to render the problem well-posed. This is shown in Figure 12.11 where for some time boundary data on the left can be prescribed when characteristics
from the boundary can be *pushed in* (see [SB06]). However when the characteristics are coming from inside, the boundary data can not be prescribed.

![Figure 12.11: Boundary Data](image)

**12.7.4 Macroscopic (PDE) Traffic Network**

The network problem for traffic flow has been studied by researchers ([GP06], [HR95], [Leb96] and [CP02]). They consider a traffic node with incoming $n$ junctions and outgoing $m$ junctions as shown in Figure 12.12.

![Figure 12.12: Traffic Node with Incoming and Outgoing Links](image)

The traffic distribution at the junction is performed based on a traffic distribution matrix that must be provided for the node as well as using an entropy condition at the node that is equivalent to maximizing the flow at the node.
We present the summary of the Coclite/Piccoli model for the network ([CP02], [GP05] and [GP06]). That summary is also used in [GHKL05]. The formulation in terms of demand and supply is shown in the work by Lebacque ([Leb96], [LK04], and [BLL96]). This formulation is equivalent to the Coclite/Piccoli formulation, and both then show numerical method using the Godunov scheme.

Each arc of the traffic network is an interval \([a_i, b_i]\). The model for the network is

\[
\frac{\partial}{\partial t} \rho^i(t, x) + \frac{\partial}{\partial x} f(\rho^i(t, x)) = 0 \quad \forall x \in [a_i, b_i], t \in [0, T] \tag{12.47}
\]

\[
\frac{\partial}{\partial t} \pi^i(t, x, k, r, s) + v^i(\rho^i(t, x)) \frac{\partial}{\partial x} \pi^i(t, x, k, r, s) = 0 \quad \forall x \in [a_i, b_i], t \in [0, T] \tag{12.48}
\]

Here \(\pi(t, x, k, r, s)\) is a function whose range is \([0, 1]\) and gives the fraction of the traffic density on path \(k\) of the OD pair \((r, s)\) on the arc \(i\). Hence, we have

\[
\rho^i(t, x, k, r, s) = \pi^i(t, x, k, r, s) \rho^i(t, x) \tag{12.49}
\]

This ensures the FIFO condition automatically since vehicle speed is a function of traffic density, and hence vehicles don’t cross each other in this model (unless we add lane modeling with lane change logic).

At any node the following flow conservation condition (Kirchoff’s law) must be satisfied. This equation says that the total inflow to a node equals its outflow.

\[
\sum_{i=1}^{n} f_i(\rho_i(b_i, t)) = \sum_{i=n+1}^{n+m} f_i(\rho_i(a_i, t)), \quad \forall t \geq 0 \tag{12.50}
\]

At the nodes, we have traffic splitting factor \(\alpha_{j,i}\) that tell us what fraction of a given incoming arc \(i\) is going to an outgoing arc \(j\) of that node. The factors \(\alpha_{j,i}\) have to be consistent with \(\pi^i(t, x, k, r, s)\).

\[
\alpha_{j,i} = \sum_r \sum_s \sum_k \pi^i(t, b_i-, k, r, s) \tag{12.51}
\]

The weak solution of the traffic density at a node is given by a collection of functions \(\rho_i\) such that the following is satisfied.

\[
\sum_{i=1}^{n+m} \int_0^\infty \int_{a_i}^{b_i} \left( \frac{\partial \phi_i}{\partial t} + f(\rho_j) \frac{\partial \phi_i}{\partial x} \right) dx dt = 0 \tag{12.52}
\]

All the details of this model can be obtained from [GP06]. The Wardrop condition for this macroscopic DTA model become the following.
(δ_{a,k}^{rs} π^i(t, a_i, k, r, s))(c_k^{rs}(t) - u_{rs}(t)) = 0
\quad c_k^{rs}(t) - u_{rs}(t) \geq 0
\quad \sum_k δ_{a,k}^{rs} π^i(t, a_i, k, r, s) = q_{rs}(t)
\quad (12.53)
\quad \sum_k δ_{a,k}^{rs} π^i(t, a_i, k, r, s) \geq 0

Here, \( i \) in the expression \( π^i(t, a_i, k, r, s) \) is the link connected to the source \( r \) for the particular \( k \) and \( s \). The travel time \( c_k^{rs}(t) \) is developed in the next section.

### 12.7.5 Travel Time Dynamics

This section provides a model for obtaining the experienced travel time function for the hydrodynamic model that can be used for the macroscopic DTA model.

Consider a link as shown in Figure 12.13. We want to develop a travel time function \( T(t, x) \) that provides the travel time for a vehicle at position \( x \) and time \( t \) to reach \( x = ℓ \). It takes a vehicle time \( \Delta x / v(t, x) \) to move from \( x \) to \( x + \Delta x \). Hence, we have the following travel time condition.

\[
T(t + \Delta t, x + \Delta x) = T(t, x) - \frac{\Delta x}{v(t, x)} \quad (12.54)
\]

Taking the Taylor series first terms for \( T(t, x) \) and simplifying, we obtain

\[
\frac{\partial T(t, x)}{\partial t} \Delta t + \frac{\partial T(t, x)}{\partial x} \Delta x = -\frac{\Delta x}{v(t, x)} \quad (12.55)
\]
CHAPTER 12. DYNAMIC TRAFFIC ASSIGNMENT: A SURVEY OF MATHEMATICAL MODELS AND TECHNIQUES

Multiplying by \( v(t, x) \), dividing by \( \Delta x \), and then taking limits and simplifying we get the travel time partial differential equation.

\[
\frac{\partial T(t, x)}{\partial t} + \frac{\partial T(t, x)}{\partial x} v(\rho(t, x)) + 1 = 0 \quad (12.56)
\]

Hence, the one-way coupled PDE system for LWR and travel time for a link is given by

\[
\frac{\partial}{\partial t} \rho(t, x) + \frac{\partial}{\partial x} [\rho(t, x) v(\rho(t, x))] = 0 \quad (12.57)
\]

\[
\frac{\partial T(t, x)}{\partial t} + \frac{\partial T(t, x)}{\partial x} v(\rho(t, x)) + 1 = 0 \quad (12.58)
\]

\[
v(\rho(t, x)) = v_f(1 - \frac{\rho}{\rho_m}) \quad (12.59)
\]

12.8 Simulation based DTA

With the availability of faster processors and computers using simulation based DTA is becoming more and more popular ([PZ01], [MHA+98], and [BABKM98]). Summary of simulation based DTA and the methodology is presented in [CBM+09] and [PZ01]. In principle, the simulation of the network can be accomplished using microscopic, mesoscopic, or macroscopic simulations. Microscopic simulation are based on car-following models and they model the vehicle dynamics for each individual vehicle. Macroscopic simulations are based on discretization and numerical solutions of the macroscopic models, such as LWR based models. Mesoscopic simulations use the fundamental relationship for obtaining vehicle speeds (macroscopic behavior), but also have individual vehicles (microscopic behavior) modeled with the tracking of their location and speeds. Since the mesoscopic modeling based DTA is more prevalent, we will focus on that in this section.

There are two main steps to prepare the simulation based DTA. A three stage iterative process to obtain user equilibrium behavior, as well as a field data based calibration process. Once these two processes have been successful, the software can be used for various studies.

12.8.1 Iterations for User Equilibrium

This equilibration process is performed in three steps ([CBM+09]). These three steps are iterated till the user equilibrium condition is obtained within some tolerance limit.
**Network Loading** This step is obtained by running the network simulation for a given time varying OD and traffic assignment to various paths between each OD pairs. The result is the set of travel times for each path.

**Path Set Update** The traffic loading obtained from the previous step is used to calculate the set of $k$-shortest paths between each OD pair.

**Path Assignment Adjustment** In this step the OD flows are assigned to new updated paths from the previous step.

### 12.8.2 Calibration from Field Data

Data obtained from field surveys and sensors can be used to calibrate the simulation based DTA models. Some parameters that can be tuned include the time varying OD values, road capacities, and vehicle speed density parameters. The calibration can be performed in order to maximize the match between the simulated outputs and the observed data. Various numerical optimization methods have been used such as gradient based methods, SPSA, etc. The general scheme is to find the parameter vector that will minimize the least squared error of the observations, where the observations are $y_i$, and the output from simulation is dependent on the parameters as $h_i(\theta)$.

$$\theta^* = \text{Arg min}_\theta \sum_i (y_i - h_i(\theta))^2$$

(12.60)

A typical iterative scheme if it is gradient based to find the optimal parameters can be

$$\theta^*[k + 1] = \theta^*[k] - \eta \nabla_\theta \sum_i (y_i - h_i(\theta))^2$$

(12.61)

OD estimation has been performed (see [BABKM98]) using an auto-regressive model for OD variations from nominal values, and then applying Kalman filter techniques on it.

### 12.9 Traffic Operation Design and Feedback Control

Traffic assignment problem and its solutions have very strong roots in the transportation planning process, especially the four-stage process shown in Section 12.1. It is very important to keep this context in mind in order to ensure its proper use. DTA models can help in performing before and after
studies for various transportation projects. They can also help in many other studies by enhancing its basic framework with additional features such as environment effects of congestion, costs etc.

For real-time traffic operations we must use and develop techniques specifically for real-time operations. For instance, if we have to design an isolated ramp control at one location, the entire OD matrix obtained and calibrated from field studies during some limited time is not relevant to that problem. Feedback control based methods are extremely suited for design of traffic control and real-time operations. The details of many specific feedback control designs for traffic operations such as real-time traffic routing and ramp metering are available in multiple publications ([KO99], [KÖ03], [KÖ98], [KÖ06], and [KÖ05]).

12.10 Conclusions

This chapter provided a mathematical survey of the static and dynamic traffic assignment problems. It presented the macroscopic DTA model using the LWR distributed parameter model as the basis. The chapter presented a new partial differential equation for travel time function for a link. It also provided a brief summary of simulation based DTA.
Appendices
SOLUTIONS TO EXERCISES

This chapter reviews …

A.1 Chapter 1 Problems
BIBLIOGRAPHY


177


**Index**

1-reducible, 105

\(O(1), 17\)
\(O(2^n), 22\)
\(O(\log(n)), 18\)
\(O(n!), 22\)
\(O(n), 18\)
\(O(n \log n), 26\)
\(O(n^2), 20, 25, 26\)
\(O(n^n), 22\)
\(O(n^p), 21\)
\(\Theta\) Notation, 14
\(\lambda\) productions, 71
\(\lambda\)-free language, 71
\(\mu\)-recursive, 98

absolutely continuous, 37
Acceptor, 49
Ackerman’s Function, 97
Addition, 96
adjacent, 128
algorithm, 93
alphabet, 45
ambiguity, 65
arc, 127
Arithmetic Expressions, 66
associative, 113
Associative Law, 114
automata, 47
Average Case, 34, 41
Best Case, 40

big-\(\Omega\), 13
Big-\(O\) Algebra, 15
Big-\(O\) Notation, 13
bijection, 79
blank tape halting, 101
Boolean Constant, 113
Boolean expression, 112
Boolean Function, 113
Boolean Satisfiability, 112
Boolean statement, 113
Boolean Variable, 113
Borel field, 36
Borel measure, 36
Boundary of a Vertex Set, 131
Bubble Sort, 20, 23

Cantor’s diagonalization method, 80
Cantor-Schröder-Bernstein Theorem, 77
Caratheodory construction, 36
cardinality, 43
certificate, 115
Chomsky Hierarchy, 86
Chomsky normal form, 67
Church Turing thesis, 94
Circuit, 134
Class L, 122
Class NL, 122
Class NP, 115
Class P, 111
clause, 115
Closure Properties, 56, 70
CNF, 67, 115
commutative, 113
Commutative Law, 114
Complexity Analysis, 40
Composition, 97
Computability, 94
computable, 95
concatenation, 44
conjunction, 112
conjuncts, 114
Context Sensitive, 85
Context-Free, 63
Context-Free Languages, 59
Cook-Levin theorem, 117
countable, 78
countably additive, 35
Cycle, 134
CYK algorithm, 67

De Morgan's Law, 114
de Morgan's law, 35
Decidability of LBA, 105
Decidable, 57
decidable, 77
Decidable Properties, 71
Decision Problem, 93
denumerable, 78
derivation, 65
derivation tree, 65
Deterministic Pushdown Automata, 67
DFA, 48
diagonalization, 80
Digraph, 128
Dirac delta, 37
discrete measure, 37
discrete random variable, 38
disjunction, 112
disjunctive normal form, 114
disjuncts, 115
distributive, 113

Distributive Law, 114
Divide and Conquer, 31
DNF, 114
doublet, 93
DSpace $(f(n))$, 120
edge, 127
effective algorithm, 94
empty, 71, 98
empty graph, 129
Empty Language Problem, 103
enumeration, 82
Equivalence, 54
Equivalent Turing Machines, 89
expectation, 37
Exponential Complexity, 22
EXPTIME, 121
Extended Church Turing thesis, 94
Factorial Complexity, 22
finite graph, 129
Finite State Automata, 48
First Theorem of Graph Theory, 134
FST, 54
Full Subtraction, 96
functional, 37
fundamental theorem of arithmetic, 79
Fundamental Theorem of Decidability, 99
Generating Functions, 42
grammar, 43, 45
Graph, 127
Graph Degrees, 132
Greibach normal form, 67
halt, 74
Halting Oracle TM, 99
Halting Problem, 98
Hypergraph, 130
Identity, 96
INDEX

impulse train, 37
incident, 128
incompressible, 107
incompressible by $\alpha$, 126
Incompressible String, 107
infinite graph, 129
inherently ambiguous, 66
Input Tape, 50
Insertion Sort, 25
isolated vertex, 128
Kleene star, 44
Kleene's Theorem, 55
Kolmogorov Complexity, 106
Labeled Graph, 130
language, 43
Language Membership Problem, 94
Lebesgue integral, 37
Lebesgue measure, 36
Lebesgue-Stieltjes integral, 37
left linear grammar, 56
line, 127
Linear Bounded Automata, 85
Linear Search, 18
literal, 115
Little-$\omega$, 14
Little-$o$ Notation, 14
LL grammar, 68
LL$(k)$ grammars, 68
log space computable function, 123
LR grammars, 68
machine, 43
Main Theorem, 34
Many-one Reduction, 105
Matrix Multiplication, 111
Mealy Machine, 52
measure, 35
measure space, 34
Median, 30
Membership for Context Sensitive
Grammar, 105
Membership Problem, 102
Merge Sort, 26
monoid, 45
Moore Machine, 53
Multidimensional Turing Machine, 91
multidigraph, 129
Multigraph, 129
Multiplication, 97
Multitape, 90
Multitape to Single Tape Simulation, 109
Multitrack, 90
Myhill-Nerode theorem, 49
ncojunctive normal form, 115
NDSPACE, 120
NDSPACE=PSPACE, 121
Neighborhood of a Vertex, 130
neighbors, 128
NL-complete, 123
nodes, 127
Non-ambiguous, 66
Noncomputable Function, 95
nondeterministic, 50
nondeterministic pushdown automata, 62
Nondeterministic to Deterministic
Turing Machine, 110
Nondeterministic Turing Machine, 91
Nonregular language, 57
Normalized, 35
NP-complete, 116
NP-hard, 116
NSPACE$(f(n))$, 120
null graph, 129
null string, 44
off-line Turing Machine, 91

187
One-way Tape, 90
Oriented Graph, 131
Output Alphabet, 54
Output Function, 54
Parsing Complexity, 67
partial function, 95
Partial Subtraction, 96
Path, 134
PATH Membership Problem, 124
points, 127
Polynomial Complexity, 21
Polynomial Time Computable Function, 116
Polynomial Time Many-one Reduction, 116
Post correspondence problem, 105
primitive recursive, 97
probability density function, 38
probability distribution, 36
probability space, 35
Productions, 46
Projection, 97
pseudodigraph, 129
Pseudograph, 129
PSPACE, 120
PSPACE Completeness, 121
PSPACE-complete, 121
PSPACE-hard, 121
Pumping lemma, 68
Pumping Lemma for Context Free Languages, 68
Pumping Lemma for Regular Languages, 58
Pushdown Automata, 59
QBF, 121
quadruple, 46
Quantified Boolean Formula, 121
Quick Sort, 26
random variables, 36
Randomizing Algorithms, 34
Read-only Tape, 91
Recursion, 97
recursive, 77
Recursive Computation, 32
recursively enumerable, 77
Reducibility, 101
Reflexivity, 17
Regular expressions, 55
Regular Grammars, 55
Regular Languages, 55
Regular Sets, 55
Rice's Theorem, 104
right linear grammar, 56
Running time, 110
s-grammar, 67
SAT, 117
Selection Sort, 24
semantics, 65
semi-decidable, 77
semigroup, 44
sequential, 65
sextuple, 54
sigma algebra, 35
simple digraph, 129
Simple Graph, 129
simulate, 89
Space Complexity, 119
Stack, 61
Start Variable, 46
state entry, 102
stochastic process, 34
string compressibility, 107
sublinear, 122
Successor, 96
Symmetry, 17
terminal symbol, 45
Terminals, 46
Time Complexity, 108
INDEX

Time Complexity Language Class, 108
Time Complexity Order, 108
total function, 95
Trail, 134
transducer, 53
transition function, 48
Transitivity, 17
Transpose Symmetry, 17
triplet, 94
trivial graph, 129
trivial property, 104
Turing Machine, 38
Turing Machine with Stay, 89
Turing Machines, 72

unique factorization, 79
unit-productions, 71
Universal Turing Machine, 93
Unrestricted Grammar, 84
useless production, 71

Variables, 46
verify, 115
vertex, 127
Vertex Degree, 132

Walk, 134
Worst Case, 40

Zero Function, 96
NOMENCLATURE

$(\Omega, \mathcal{G}, P)$ Probability Space

$2^A$ Power set of a set $A$, i.e. the set of all subsets of $A$

$\Delta(\mathcal{G})$ Maximum degree of a graph

$\delta(\mathcal{G})$ Minimum degree of a graph

$\ell(\omega)$ Length of a string: number of symbols in the string

$\lambda$ Null symbol/string

$\lceil f \rceil$ Roof function giving the least integer whose value is greater than or equal to $f$

$\lfloor f \rfloor$ Floor function giving the greatest integer whose value is less than or equal to $f$

$\mathbb{N}$ The set of natural numbers

$\mathbb{Q}$ The set of rational numbers

$\mathbb{R}$ The set of real numbers

$\mathbb{Z}$ The set of integers

$\mathcal{L}$ A specific Turing semidecidable language which is not Turing decidable

$\mathcal{P}$ Problem set for a decision problem

$\mathcal{A}$ Algorithm

$\mathcal{E}$ Evaluation function for a decision problem

$\mathcal{G} = (\mathcal{V}, \mathcal{E})$ Graph $\mathcal{G}$ with vertex set $\mathcal{V}(\mathcal{G})$ and edge set $\mathcal{E}(\mathcal{G})$

$\mathcal{S}$ Sigma Algebra of events

$\mathcal{T}_m$ Modified Turing Machine

$\Omega$ Probability Universal Set: Set of all outcomes

$\Omega(f)$ Big Omega notation

$\omega(o)$ Little Omega notation
INDEX

\( \phi \)  Empty set
\( \Theta(f) \)  Theta notation
\( A \leq_m B \)  Language \( A \) is \( m \)-reducible to language \( B \)
\( A^* \)  Kleene star of a set, i.e. a set of all strings obtained from the alphabet set \( A \).
\( A^c \)  Complement of set \( A \)
\( N(\Omega) \)  Open neighborhood of a set of vertices \( v \) in a graph.
\( N(v) \)  Open neighborhood of a vertex \( v \) in a graph.
\( N[\Omega] \)  Closed neighborhood of a set of vertices \( v \) in a graph.
\( N[v] \)  Closed neighborhood of a vertex \( v \) in a graph.
\( O(f) \)  Big Oh notation
\( o(f) \)  Little Oh notation

3-SAT  Boolean Satisfiability Problem or Language with three literals in every clause

Class NP  Class with polynomial time verification complexity
Class NP-complete  NP problems, such that every NP problem is polynomially reducible to the problem
Class NP-hard  Any problem, such that every NP problem is polynomially reducible to the problem

Class P  Class with polynomial time complexity
\( \deg(v) \)  Degree of a vertex in a graph
\( \text{DSPACE}(f(n)) \)  Language decidable by a deterministic Turing machine of space complexity of order \( O(f(n)) \)
\( \text{NSPACE}(f(n)) \)  Language decidable by a nondeterministic Turing machine of space complexity of order \( O(f(n)) \)
\( \text{NTIME}(n^i) \)  Class with time complexity for Verification of order \( n^i \)
\( \text{PSPACE}(f(n)) \)  Language decidable by a deterministic Turing machine of space complexity of polynomial order
QBF  Quantified Boolean Formula
SAT  Boolean Satisfiability Problem or Language
\( \text{TIME}(n^i) \)  Class with time complexity for Decidability of order \( n^i \)