Algorithms for Problems in Voting and Scheduling

by

Ehsan Iranmanesh

M.Sc., Amirkabir University of Technology, 2008
B.Sc., Shahid Bahonar University of Kerman, 2005

Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy

in the
School of Computing Science
Faculty of Applied Sciences

© Ehsan Iranmanesh 2016
SIMON FRASER UNIVERSITY
Fall 2016

All rights reserved.
However, in accordance with the Copyright Act of Canada, this work may be reproduced without authorization under the conditions for “Fair Dealing.” Therefore, limited reproduction of this work for the purposes of private study, research, education, satire, parody, criticism, review and news reporting is likely to be in accordance with the law, particularly if cited appropriately.
Approval

Name: Ehsan Iranmanesh
Degree: Doctor of Philosophy (Computing Science)
Title: Algorithms for Problems in Voting and Scheduling
Examinig Committee: Chair: Dr. Steven Pearce

Dr. Ramesh Krishnamurti
Senior Supervisor
Professor

Dr. Binay Bhattacharya
Supervisor
Professor

Dr. Oliver Schulte
Internal Examiner
Professor

Dr. Robert Benkoczi
External Examiner
Associate Professor
Department of Mathematics and Computer Science
University of Lethbridge

Date Defended: 24 November 2016
Abstract

In this dissertation, we study the voting problem and the ranking problem in computational social choice, as well as a matching problem in a restricted graph. We present our results for these problems in two parts.

Part I: Election, Ranking, and Heuristics. Computational voting theory is an interdisciplinary research area which considers the computational problems that arise in voting. Selecting the winner(s) of an election is one such problem. The problem of computing the winner(s) using most voting rules is easy. However, there are a few rules for which this problem becomes computationally hard. We study two important voting and ranking rules under which computing the winner(s) is hard. The first voting procedure is the Chamberlin-Courant which is a proportional representation system that does not restrict candidates to have a minimum number of votes to be selected in an assembly. We consider domination analysis of a 2-Opt heuristic for the winner determination problem under the Chamberlin-Courant system. The next problem we consider is the Linear Ordering Problem. Linear ordering problem is a classic optimization problem which can be used to model problems in graph theory, machine scheduling, and ranking (also known as Kemeny rule under social choice domain). Relatively recently, there has been some success in using Mixed Integer Program (MIP) heuristic for NP-hard optimization problems. We report our experience with using a MIP heuristic for the problem.

Part II: Matching. The first problem we consider in this part is the Linear Ordering Problem. We show how the linear program of this problem can be solved by using a primal-dual based combinatorial algorithm instead of the Simplex method. Next, we address the cyclical scheduling problem which is used to schedule shifts for workers in a factory. Given a set of $n$ work periods, each worker is assigned a shift where he works for $n - 2$ consecutive periods and takes off the remaining two periods. Each shift may also have a cost associated with it. We use the primal-dual method to solve $(n - 2, n)$ cyclical scheduling problem by solving a series of $b$-matching problems on a cycle of $n$ vertices.

Keywords: Computational Social Choice, Proportional Representation, Chamberlin-Courant System, Linear Ordering Problem, Kemeny Rule, Mixed Integer Program Heuristic, $b$-matching, Cyclical Shifts, Scheduling
To my love, Maryam

To my parents, for their endless love and support
I would like to express my special appreciation and thanks to my advisor Dr. Ramesh Krishnamurti. It was a great experience working with Ramesh. Not only did I learn a lot of things about algorithm and optimization from him, but he also taught me invaluable life lessons. He is extremely nice, kind, and patient, and I would never have been able to finish my dissertation without his guidance and encouragement. I would like to sincerely thank Dr. Binay Bhattacharya for his support, guidance, and encouragement. He is very nice and it was always delightful to discuss research and no-research topics with him. I would like to give my regards and appreciations to Dr. Abraham Punnen for his helpful insights and suggestions. I really appreciate his encouragement and valuable comments. I would also like to thank Dr. Oliver Schulte and Dr. Robert Benkoczi for taking the time and effort to read my thesis and giving me very useful comments.

I am forever indebted to all my co-authors, the results in this thesis were based on joint work with Ramesh Krishnamurti, Binay Bhattacharya, and Ante Ćustić. I thank them for all the things they taught me. I also thank Elsevier for permission to include Chapter 5 of my dissertation, which was originally published in Theoretical Computer Science journal.

I thank my fellow lab-mates and friends: Kamyar, Arash, Hossein, Carrie, and Hedayat in the theory lab for the stimulating discussions and for all the fun we have had in the last few years. My friends at Simon Fraser University make my studies more pleasant and enjoyable. Especially, I am grateful to Majid, Hassan, Alireza, Mohammad, Alireza, Hamed, Bahareh, Amin, and Mohsen.

Heartfelt thanks goes to my family, words cannot express how grateful I am to my parents for all of the sacrifices that they have made on my behalf. They are always my sense of safety and calmness. Thank you for giving me everything in life and for being so loving, caring and supportive.

Last but not least, I am immensely grateful to my wife, Maryam, for all her love, patience and kindness. Her determination, confidence, and courage was inspiring for me, I was very fortunate to have her as a lab-mate and wife in my life and I could not have completed this work without her unfaltering love and support.
# Table of Contents

Approval .................................................. ii  
Abstract .................................................. iii  
Dedication .................................................. iv  
Acknowledgements ....................................... v  
Table of Contents ....................................... vi  
List of Tables ............................................ ix  
List of Figures ........................................... x  

## 1 Introduction  
1.1 Computationally Hard Voting Rules  ................................... 2  
1.1.1 Dodgson Rule ........................................... 2  
1.1.2 Young Rule ............................................ 3  
1.1.3 Kemeny Rule ........................................... 4  
1.1.4 Slater Rule ............................................ 4  
1.1.5 Proportional Representation  ................................ 5  
1.2 Structured Preferences ................................... 7  
1.2.1 Single-Peaked .......................................... 7  
1.2.2 Single-Crossing ....................................... 7  
1.3 Structure and Overview of Results  ............................. 8  
1.4 Bibliographic Notes ..................................... 8  

## 2 2-Opt Heuristic for the Winner Determination Problem under the Chamberlin-Courant System  
2.1 Introduction ............................................ 9  
2.2 Integer Linear Programming Formulation for WDPCC  ............... 11  
2.3 2-Opt Heuristic ....................................... 12
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3.1 Computing the Average Solution Value</td>
<td>13</td>
</tr>
<tr>
<td>2.3.2 Computing the Domination Number</td>
<td>13</td>
</tr>
<tr>
<td>2.3.3 Domination Analysis of 2-Opt</td>
<td>16</td>
</tr>
<tr>
<td>2.3.4 Empirical Analysis</td>
<td>18</td>
</tr>
<tr>
<td>3 Heuristic for Linear Ordering Problem</td>
<td>20</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>20</td>
</tr>
<tr>
<td>3.2 Problem Formulations</td>
<td>22</td>
</tr>
<tr>
<td>3.3 Methods Used</td>
<td>23</td>
</tr>
<tr>
<td>3.3.1 LP-Based Heuristic</td>
<td>23</td>
</tr>
<tr>
<td>3.3.2 Mixed Integer Program Heuristic</td>
<td>24</td>
</tr>
<tr>
<td>3.3.3 Alternative Algorithm for Starting Solution</td>
<td>25</td>
</tr>
<tr>
<td>3.4 Empirical Analysis</td>
<td>27</td>
</tr>
<tr>
<td>3.4.1 Data Set</td>
<td>27</td>
</tr>
<tr>
<td>3.4.2 Computational Results</td>
<td>28</td>
</tr>
<tr>
<td>4 Combinatorial Algorithm for Solving Linear Program of Linear ordering Problem</td>
<td>33</td>
</tr>
<tr>
<td>4.1 Introduction</td>
<td>33</td>
</tr>
<tr>
<td>4.2 Primal-Dual Algorithm</td>
<td>34</td>
</tr>
<tr>
<td>4.3 Graphical Representation of DRP</td>
<td>37</td>
</tr>
<tr>
<td>4.3.1 Definitions</td>
<td>38</td>
</tr>
<tr>
<td>4.3.2 Combinatorial Algorithm for Linear Program of Linear Ordering Problem</td>
<td>43</td>
</tr>
<tr>
<td>5 Cyclical Shift Scheduling</td>
<td>54</td>
</tr>
<tr>
<td>5.1 Introduction</td>
<td>54</td>
</tr>
<tr>
<td>5.2 The $b$-Matching Formulation</td>
<td>55</td>
</tr>
<tr>
<td>5.3 Definitions</td>
<td>57</td>
</tr>
<tr>
<td>5.4 A Simple Augmentation Algorithm</td>
<td>58</td>
</tr>
<tr>
<td>5.5 Properties of Augmenting Paths</td>
<td>60</td>
</tr>
<tr>
<td>5.6 Algorithm Efficient Largest-cost Augmentation</td>
<td>63</td>
</tr>
<tr>
<td>5.6.1 Description and Running Time of Algorithm Efficient Largest-cost Augmentation</td>
<td>63</td>
</tr>
<tr>
<td>5.6.2 Running Time of Algorithm Efficient Large-cost Augmentation</td>
<td>66</td>
</tr>
<tr>
<td>5.7 Details of Algorithm Augment-Interval</td>
<td>66</td>
</tr>
<tr>
<td>5.7.1 Detailed Description</td>
<td>67</td>
</tr>
<tr>
<td>5.7.2 Constructing the Solution</td>
<td>70</td>
</tr>
<tr>
<td>6 Conclusions</td>
<td>74</td>
</tr>
<tr>
<td>6.1 Winner Determination Problem and Heuristics</td>
<td>74</td>
</tr>
</tbody>
</table>
List of Tables

Table 3.1  Computational results for instances RandAI  . . . . . . . . . . . . . . . . . . .  30
Table 3.2  Computational results for instances RandAII  . . . . . . . . . . . . . . . . .  31
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Example of an equivalence class defined by $\sim_i$, for $\mathcal{A}_i = {{1, 2, 3}, {4, 5, 6}}$ and $\pi_i$ defined by: $\pi_i(j) = j$ for $j \leq 3$, $\pi_i(j) = j - 3$ for $j &gt; 3$.</td>
</tr>
<tr>
<td>2.2</td>
<td>Number of voters is increasing, number of candidates and $k$ are fixed.</td>
</tr>
<tr>
<td>2.3</td>
<td>Number of voters and candidates are increasing, $k$ is fixed.</td>
</tr>
<tr>
<td>2.4</td>
<td>Number of voters, candidates and $k$ are increasing.</td>
</tr>
<tr>
<td>3.1</td>
<td>Objective function value box-plot of LP versus GRASP.</td>
</tr>
<tr>
<td>3.2</td>
<td>CPLEX integrality gap vs. MIP heuristic integrality gap.</td>
</tr>
<tr>
<td>4.1</td>
<td>Example of DRP graph.</td>
</tr>
<tr>
<td>4.2</td>
<td>Example of augmenting paths.</td>
</tr>
<tr>
<td>4.3</td>
<td>Example of semi-augmenting path.</td>
</tr>
<tr>
<td>4.4</td>
<td>Example of augmenting lasso of Type I.</td>
</tr>
<tr>
<td>4.5</td>
<td>Example of augmenting lasso of Type II.</td>
</tr>
<tr>
<td>4.6</td>
<td>Example of augmenting semi-lasso of Type I.</td>
</tr>
<tr>
<td>4.7</td>
<td>Example of augmenting semi-lasso of Type II.</td>
</tr>
<tr>
<td>4.8</td>
<td>Subgraph corresponds to constraints for $y_1, y_2$, and $y_3$.</td>
</tr>
<tr>
<td>5.1</td>
<td>Example of odd length augmenting path of Type I.</td>
</tr>
<tr>
<td>5.2</td>
<td>Example of even length augmenting path of Type I.</td>
</tr>
<tr>
<td>5.3</td>
<td>Example of odd length augmenting path of Type II.</td>
</tr>
<tr>
<td>5.4</td>
<td>Example of even length augmenting path of Type II.</td>
</tr>
<tr>
<td>5.5</td>
<td>Initial state.</td>
</tr>
<tr>
<td>5.6</td>
<td>Path $e_4 = {v_4, v_5}$.</td>
</tr>
<tr>
<td>5.7</td>
<td>Path $e_6 = {v_6, v_0}$.</td>
</tr>
<tr>
<td>5.8</td>
<td>Path $e_5 = {v_5, v_6}$.</td>
</tr>
<tr>
<td>5.9</td>
<td>Path $e_1 = {v_1, v_2}$.</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Social choice theory is the study of procedures for making collective choices that affect a group of people. It basically considers the following question: how should we aggregate the preferences of a group of agents to obtain a social preference? Social choice theory traditionally was studied in economics and political science [57]. Recently, computational social choice has emerged as an interdisciplinary research area at the intersection of economics, political science, computer science, operations research and computational logic. The relationship between social choice theory and computer science can be seen as a two-way street. Social choice theory uses the notions and methods in computer science for solving problems that arise in the area. Some examples include formal specification and verification of social procedures using mathematical logic, using techniques in artificial intelligence and logic to compactly represent preference in the combinatorial domain, deciding the winner(s) in voting procedures and finding some voting rules which make cheating computationally hard.

Computer science has also borrowed concepts and methods from social choice theory for solving problems. Examples include managing societies of rational and autonomous agents using negotiation techniques and voting procedures and application of social choice theory techniques to develop page rank systems for internet search engines [26].

Research in the computational social choice can be divided into the following topics:

- **Voting**: is one of the most popular ways to reach a common decision. The problem of computing the winner(s) using most voting rules is easy. However, there are a few rules for which this problem becomes computationally hard. We consider some of these rules in Section 1.1.

- **Fair allocation**: deals with the problem of assigning a set of items $I$ to a set of agents $A$, while agents have preferences over all possible combinations of items

- **Coalition formation**: deals with agent cooperation in order to achieve common goals
Our focus in the first part of this dissertation is on computationally hard aggregation rules.

1.1 Computationally Hard Voting Rules

We need some notations and definitions. Throughout this chapter, we refer to the set of voters by $V$ and the set of candidates by $C$, where $|V| = n$, and $|C| = m$. Whenever we talk about an election, there is a set of voters $V$, a set of candidates $C$, the voters either rank the candidates or give them some scores, which implicitly implies their preferences over the candidate set, and the goal is to select winner(s) that satisfies voters the most. Pairwise majority contest compares each pair of candidates head-to-head, and award each candidate one point for each one-on-one victory.

**Definition 1.1.1 ([62]).** A candidate that beats every other candidate in pairwise majority contests is called a Condorcet winner.

Such a winner might not always exist, but whenever a Condorcet winner exists, it must be unique. Likewise, we can define the Condorcet loser as a candidate that is defeated by every other candidate in pairwise majority contests.

**Definition 1.1.2 ([62]).** A voting rule satisfies the Condorcet principle if it elects only the Condorcet winner whenever one exists.

Next, we give a brief overview of some voting rules which always select the Condorcet winner if such a candidate exist.

1.1.1 Dodgson Rule

The idea behind the Dodgson rule, which was proposed in 1867 by Dodgson, is to select the Condorcet winner if it exists. However, if there is none, a candidate who is the most similar one to the Condorcet winner should be chosen. The notion of similarity to the Condorcet winner is defined as follows. Suppose it is possible to change the ranking of any voter through the pairwise interchange of candidates adjacent in the voter’s preference order. Then a candidate who can be the Condorcet winner by the minimum number of preference changes of the voters is the Dodgson winner [72, 26]. These shifts in the voter’s preference order are called switches. It is evident that when the Condorcet winner exists, no switches are needed. The Dodgson score of a candidate is the minimum number of switches in the voter’s preference that make that candidate the Condorcet winner. Computing the Dodgson score of a candidate can be cast as a linear-integer program. Assume we want to compute the Dodgson score of candidate $a^*$. Let $def(a)$ be number of additional voters that must rank $a^*$ above $a$ in order for $a^*$ to beat $a$. Binary variable $x_j^i$ is defined as follows ($J = \{0, 1, 2, \ldots, m - 1\}$):

$$x_j^i = \begin{cases} 1, & \text{iff } a^* \text{ is moved upward by } j \text{ positions in the ranking of voter } i \\ 0, & \text{otherwise} \end{cases}$$
and:
\[
e_{ja}^i = \begin{cases} 
1, & \text{iff pushing } a^\ast \text{ by } j \text{ positions in the ranking of voter } i \\
\text{makes } a^\ast \text{ gain additional votes against } a \\
0, & \text{otherwise}
\end{cases}
\]

Then the linear-integer program formulation to compute the Dodgson score of \( a^\ast \) would be:
\[
\begin{align*}
\text{Min} & \quad \sum_{i \in V, j \in J} jx_j^i \\
\text{s.t.} & \quad \sum_{j \in J} x_j^i = 1 \quad \forall i \in V \\
& \quad \sum_{i \in V, j \in J} x_j^i e_{ja}^i \geq \text{def}(a) \quad \forall a \in C - \{a^\ast\} \\
& \quad x_j^i \in \{0, 1\} \quad \forall i \in V, \forall j \in J
\end{align*}
\]

The winner determination problem under the Dodgson rule is hard, and the complexity of this problem is addressed in [11, 48]. Several algorithms have been proposed to approximate the Dodgson winner in the literature [51, 69, 23].

### 1.1.2 Young Rule

The Young procedure is similar to the Dodgson rule in that the Young score of a candidate \( a \) is defined as the size of the largest subset of voters for which \( a \) is the Condorcet winner (alternatively, the smallest number of voters whose removal makes \( a \) the Condorcet winner) [75, 26]. The winner is a candidate who has the highest Young score.

The Young score for each candidate can be obtained by solving an integer program. Procaccia [75] formulate the Young score problem as an integer program. Let \( a^\ast \) be the candidate whose Young score we wish to compute, and let \( x_i \) be the decision variable for each voter, defined as follows:
\[
x_i = \begin{cases} 
1, & \text{iff voter } i \text{ is included in the subset of voters for } a^\ast \\
0, & \text{otherwise}
\end{cases}
\]

There is also a constant \( e_i^a \) with the following definition:
\[
e_i^a = \begin{cases} 
1, & \text{iff voter } i \text{ prefers } a^\ast \text{ to } a \\
-1, & \text{otherwise}
\end{cases}
\]

Then we can obtain,
\[
\begin{align*}
\text{Max} & \sum_{i \in V} x_i \\
\text{s.t.} & \sum_{i \in V} x_i e_i^a \geq 1 \quad \forall a \in C - \{a^*\} \\
& x_i \in \{0, 1\} \quad \forall i \in V
\end{align*}
\]

(1.5) (1.6) (1.7)

Objective function (1.5) maximizes the number of voters we want to keep in the election. Constraint (1.6) ensures that \(a^*\) beats all the other candidates when we restrict the set of voters to the ones we keep in the election. Procaccia [75] proves that the Young score is inapproximable, and Caragiannis et al. [23] provide an almost complete treatment of the approximability of the Young rule.

### 1.1.3 Kemeny Rule

Kemeny rule aggregates \(n\) voter rankings into a collective ranking (Kemeny consensus) which is as close as possible to the \(n\) voter rankings [58, 59]. Computing a Kemeny consensus is NP-hard [11]. The practical aspects of computing the Kemeny consensus is addressed in [35, 30]. Other works have focused on approximations of the Kemeny rule in polynomial time [2]. This problem is also known as the linear ordering problem outside the social choice community. We consider this problem under this name (linear ordering problem) in Chapters 3 and 4.

### 1.1.4 Slater Rule

Slater rule [84], like Kemeny rule, aggregates \(n\) voter rankings into a collective ranking (Slater ranking) minimizing the number of ordered pairs \((a, b)\) such that \(a \succ b\) but \(b\) defeats \(a\) in their pairwise election [26, 28].

The pairwise election graph \(G(C, E)\), introduced in [31], is a graph where its vertex set corresponds to candidates and there is an edge from candidate \(a\) to \(b\) if \(a\) is the winner of a pairwise election (there is no edge between \(a\) and \(b\) if they are tied). The final Slater ranking is an acyclic tournament (an oriented complete graph) that minimizes the number of inverted edges [31].

Let

\[
x_{ab} = \begin{cases} 
1, & \text{if } a \text{ is ranked below } b \text{ in the aggregate ranking} \\
0, & \text{otherwise}
\end{cases}
\]

The integer program formulation for finding the Slater ranking is obtained as follows [31]:
The above integer program can be interpreted as a selection of the minimum number of edges whose removal makes the graph an acyclic tournament. Finding the Slater ranking is NP-hard [2, 29, 5, 53], and computational aspects of finding the Slater ranking is addressed by Charon, Hudry, and Conitzer [25, 52, 29]. Slater rule can be seen as the unweighted version of Kemeny rule.

### 1.1.5 Proportional Representation

Most of the voting rules that we consider until now elect only one single winner. However, some of them, like the Kemeny rule, can be used to elect multiple winners. Here, we present some rules that can be used in multiple winner elections. A basic property that one can expect from such a voting rule is Proportional Representation (PR). A voting rule is called PR if the power of candidates in the selected assembly is proportionate to the number of votes received under that rule. Here we briefly analyze two different rules which satisfy PR.

**Monroe Rule**

The first PR system was proposed by Monroe [71] in 1995. The Monroe rule selects as the winners those candidates which minimize the sum of the misrepresentation values of the voters. The misrepresentation values are obtained from the profiles of the voters. Assume each voter ranks all candidates and \( \mu_{ij} \) shows the rank of candidate \( j \) on the profile of voter \( i \). Then the misrepresentation value \( r_{ij} \) is defined as \( \mu_{ij} - 1 \). The winner determination problem under Monroe rule can be formulated as an integer program with two sets of decision variables. The first set of variables is \( x_{ij} \) and defined as follows:

\[
x_{ij} = \begin{cases} 
1, & \text{if candidate } j \text{ is assigned to voter } i \\
0, & \text{otherwise}
\end{cases}
\]

The next set is \( y_j \).
\[ y_j = \begin{cases} 
1, & \text{if candidate } j \text{ is selected} \\
0, & \text{otherwise}
\end{cases} \]

Using these sets of decision variables, the integer program formulation is [18]:

\[
\begin{align*}
\min & \sum_{i \in V} \sum_{j \in C} r_{ij} x_{ij} \\
\text{s.t.} & \quad \sum_{j \in C} y_j = k \\
& \quad \sum_{j \in C} x_{ij} = 1 \quad \forall i \in V \\
& \quad -L y_j + \sum_{i \in V} x_{ij} \geq 0 \quad \forall j \in C \\
& \quad -U y_j + \sum_{i \in V} x_{ij} \leq 0 \quad \forall j \in C \\
& \quad x_{ij} \in \{0, 1\} \quad \forall i \in V, \forall j \in C \\
& \quad y_j \in \{0, 1\} \quad \forall j \in C
\end{align*}
\]

The objective function (1.13) minimizes the misrepresentation value. The constraint (1.14) ensures that exactly \( k \) candidates are chosen as winners. Constraint (1.15) is to ensure that exactly one candidate is assigned to each voter. Constraints (1.16) and (1.17) ensure that the number of voters assigned to each candidate is greater than \( L \) and less than \( U \). These constraints aim to balance the number of votes each candidate receives (\( L = \lfloor \frac{n}{k} \rfloor \) and \( U = \lceil \frac{n}{k} \rceil \)). Constraints (1.18) and (1.19) are integrality constraints.

The decision version of the Winner Determination Problem (WDP) under Monroe rule is given below.

**Definition 1.1.3.** In the Winner Determination Decision Problem (WDDP), we are given the set of voters \( V \), the set of candidates \( C \), integers \( k, t \in V \) and misrepresentation values \( r_{ij} \in \mathbb{R}^+ \). The problem is to determine whether there exists a subset \( S \subseteq C \) such that \( |S| = k \), with misrepresentation at most \( t \).

Procaccia [74] showed the WDDP under the Monroe rule is NP-complete. He also provided a polynomial time algorithm for the WDP when \( k \) is constant.
The Chamberlin-Courant System

The Chamberlin-Courant rule [24] is very similar to the Monroe rule except that it is not necessary for the winners to be assigned an equal number of voters. Formally, Constraint (1.16) is dropped and $U = n$ in Constraint (1.17). Under the Monroe rule, all the winners are assigned to an equal number of voters. Thus, each winner has the same power or importance in the assembly. This is not true under the Chamberlin-Courant rule. Instead, a weighted voting may be used to reflect a winner’s power in the assembly. The number of voters $\sum_i x_{ij}$, assigned to candidate $j$, can be used as a weight for that candidate. Procaccia [74] also showed the WDDP under the Chamberlin-Courant rule is also NP-complete.

1.2 Structured Preferences

Restricted domains of voter preferences is an active research area in social choice, where voter preferences are only allowed to have a certain structure. We next discuss two such structures which are well-studied in computational social choice. In the previous section, we introduced some voting procedures for which computing the winner(s) is computationally hard. Winner determination problem for some of these voting procedures becomes computationally easy under these restrictions.

1.2.1 Single-Peaked

A preference profile is single-peaked if there exists a complete ordering of the candidates such that any voter’s preference in this ordering is either

- always strictly increasing, or
- always strictly decreasing, or
- first strictly increasing and then strictly decreasing

The single-peaked property can be generalized to trees. A profile is single-peaked on a tree if we can construct a tree whose vertices are the candidates, and for any path in this tree, the ordering of candidates with respect to that path has the single-peaked property [37].

1.2.2 Single-Crossing

A preference profile is single-crossing if there exists a complete ordering of the voters such that for any pair of candidates $a$ and $b$ in this ordering, either

- all voters $a < b$, or
- all voters $b < a$, or
• there is a single point in the ordering where the voters switch from preferring one candidate to the other one

Similar to the single-peaked property, single-crossing can also be generalized to trees. A profile is single-crossing on a tree if we can construct a tree whose vertices are the candidates, and for any path in this tree, the ordering of candidates with respect to that path has the single-crossing property [27].

1.3 Structure and Overview of Results

In Chapter 2, we design a heuristic for the winner determination problem under Chamberlin-Courant system, and theoretically analyze the heuristic. In Chapter 3, we design and implement two mixed integer program heuristics for the linear ordering problem, and experimentally analyze these heuristics against an exact mixed integer program solver. Chapter 4 is again devoted to the linear ordering problem. We show how the linear program of this problem can be solved by using a primal-dual based combinatorial algorithm instead of the Simplex method. In Chapter 5, we address the cyclical scheduling problem which is used to schedule shifts for workers in a factory. We use the primal-dual method to solve the \((n - 2, n)\) cyclical scheduling problem by solving a series of \(b\)-matching problems on a cycle of \(n\) vertices.

1.4 Bibliographic Notes

Chapter 2 is based on joint work with Ante Ćustić and Ramesh Krishnamurti [33]. Chapters 3 and 4 are based on joint work with Ramesh Krishnamurti [54]. Chapter 5 is based on joint work with Binay Bhattacharya, Soudipta Chakrabortyi, and Ramesh Krishnamurti [16].
Chapter 2

2-Opt Heuristic for the Winner Determination Problem under the Chamberlin-Courant System

2.1 Introduction

Proportional representation voting arises in voting procedures which elect multiple winners in an election. Such procedures can be used to elect any representative body comprising more than one individual, such as a committee, a council, or a legislative assembly. Informally, in this voting procedure, the size of the subset of candidates elected as winners is proportional to the number of voters that prefer them. Thus, special interest groups gain representation in proportion to their size [18]. Brams and Fishburn [19, 20] first proposed proportional representation as a voting procedure. Choosing a set of candidates that minimizes the sum of the misrepresentation values of voters was proposed by Monroe [71]. In this system, for each voter, the 'misrepresentation' of the candidate (in the chosen subset) that best represents the voter is counted. The objective is to minimize the sum of the misrepresentation count over all the voters. This rule selects as winners a subset of candidates that minimizes the sum of the misrepresentation values of the voters. The misrepresentation values are obtained from the profiles of the voters. Each voter ranks all the candidates in her profile. If $\mu_{ij}$ is the rank of a candidate $j$ in voter $i$’s profile, then the misrepresentation of that candidate for the voter is defined to be $\mu_{ij} - 1$ (the candidate with rank 1 has a misrepresentation of 0). Monroe also recommended approval balloting, a procedure similar to approval voting, where each voter approves as many alternatives as she wishes. In this paper, we study a restricted version of Monroe’s voting procedure, called the Chamberlin-Courant rule, proposed by Chamberlin and Courant [55]. While Monroe’s rule imposes upper and lower limits on the number of votes assigned to a chosen candidate, the Chamberlin-Courant system imposes no such limits. We call these versions of the
problem the 'minisum' versions. Procaccia et al. [75] show that the minisum versions for both these voting procedures are computationally intractable in general. They also provide a polynomial-time algorithm (polynomial in $m$) for the problem, where the degree of the polynomial is proportional to the size of the assembly. Betzler, Slinko and Uhmann [15] propose variants of the problem where the objective is to minimize the maximal misrepresentation. The 'minimax' versions of the problem continue to be NP-Hard for both the Monroe rule and the Chamberlin-Courant rule. Betzler et al. also provide fixed-parameter tractability results for both the minisum and the minimax versions under the Monroe system and the Chamberlin-Courant system. They also provide polynomial algorithms for the problem when there is an ordering of candidates such that the ranking in each voter’s profile is single-peaked. Skowron, Yu, Faliszewski and Elkind [83] consider the complexity of the problem under the single-crossing assumption. Skowron et al. provide a polynomial algorithm for this version under the Chamberlin-Courant system, and show its intractability under the Monroe system. Yu, Chan and Elkind [87] consider the complexity of the problem for elections that are single-peaked on a tree. Yu et al. show that the minimax version of the Chamberlin-Courant system is polynomially solvable for any tree and any misrepresentation function under this assumption. For the minisum version, their algorithm runs in polynomial time (polynomial in $n, m$, and $k$) when the tree has a small number of leaves. Clearwater, Puppe and Slinko [27] consider the single-crossing on a tree assumption and show that both minimax and minisum versions of the Chamberlin-Courant system are polynomially solvable under this assumption. Skowron et al. [82] show that no constant-factor approximation algorithms exist for the minisum version under both the Monroe system and the Chamberlin-Courant system. In work more related to our work, Skowron, Faliszewski and Slinko [81] provide experimental results for several simple and fast heuristics that obtain near optimal solutions. Here we consider the minisum version of the Winner Determination Problem under Chamberlin-Courant system (WDPCC). We are given a set of $n$ voters ($V$), and $m$ candidates ($C$). Each voter ranks candidates, giving them a distinct number between 1 and $m$ ($\mu_{ij}$ is the ranking of voter $i$ for candidate $j$, and the misrepresentation value of voter $i$ for candidate $j$ is $r_{ij} = \mu_{ij} - 1$). Let $\mathcal{S}_k = \{S \subseteq C : |S| = k\}$. For each $S \in \mathcal{S}_k$, we can define an assignment function, $\mathcal{A}_S$, that assigns each voter to a candidate in $S$ ($\mathcal{A}_S : V \rightarrow S$). Under assignment function $\mathcal{A}_S$, the misrepresentation value of voter $i$ is $r_{i\mathcal{A}_S(i)}$, and the sum of misrepresentation values is $\sum_{i \in V} r_{i\mathcal{A}_S(i)}$. The objective under WDPCC is to select $S \in \mathcal{S}_k$ and $\mathcal{A}_S$ that minimizes the sum of misrepresentation values.

We design a heuristic for WDPCC. Different metrics in the literature are used to measure the quality of a heuristic. This includes worst case analysis [86], domination ratio [6, 47], domination number [6, 47, 88, 79, 78], and comparison to average solution value [79, 7, 78].

In this paper we focus on comparison to average solution value and domination number. Let $x^{\Phi} \in \mathcal{F}(I)$ be a solution provided by algorithm $\Phi$ on a given instance $I$ of an optimization problem ($\mathcal{F}(I)$ denotes the set of feasible solutions for the instance $I$). Let $\mathcal{F}^{\Phi}(I) = \{x \in \mathcal{F}(I) : f(x) \geq f(x^{\Phi})\}$ where $f(x)$ denotes the objective function value of $x$. Furthermore, let $\mathcal{F}$ be the collection
of all instances of the problem with a fixed instance size. Then

\[ \inf_{I \in \mathcal{J}} |\mathcal{G}\Phi(I)| \text{ and } \inf_{I \in \mathcal{J}} \frac{|\mathcal{G}\Phi(I)|}{|\mathcal{F}(I)|}, \]

are called \textit{domination number} and \textit{domination ratio} of \( \Phi \), respectively \([6, 47]\).

We first compute the average solution value of the WDPCC. We show that any solution to WDPCC with an objective function value less than the average value dominates \( \binom{m-1}{k-1}k^{n-1} \) solutions (\( n \) is the number of voters, \( m \) the number of candidates, and \( k \) the number of candidates that needs to be selected). We then design a simple heuristic to obtain a solution better than the average in polynomial time. Hence, we present a heuristic with domination number no worse than \( \binom{m-1}{k-1}k^{n-1} \), and domination ratio no worse than \( \frac{1}{m} \).

2.2 Integer Linear Programming Formulation for WDPCC

The first integer programming formulation for this system under both the Monroe rule and the Chamberlin-Courant rule was given by Potthof and Brams \([17]\). Each voter provides a ranking of the candidates in set \( C \) in her profile, i.e. assigns a distinct integer from 1 to \( m \) (\( |C| = m \)) to each of the candidates. We let \( \mu_{ij} \) denote the rank voter \( i \) assigns to candidate \( j \), and \( r_{ij} = \mu_{ij} - 1 \) denote the misrepresentation of voter \( i \) by candidate \( j \).

There are two sets of decision variables in the formulation: \( x_{ij} \) and \( y_j \), where \( x_{ij} = 1 \) if candidate \( j \) is assigned to voter \( i \) \((x_{ij} = 0 \text{ otherwise})\), and \( y_j = 1 \) if candidate \( j \) is chosen \((y_j = 0 \text{ otherwise})\).

Minimize \( \sum_{i \in V} \sum_{j \in C} r_{ij}x_{ij} \) \hspace{1cm} (2.1)

s.t. \( \sum_{j \in C} y_j = k \) \hspace{1cm} (2.2)

\( \sum_{j \in C} x_{ij} = 1 \) \hspace{1cm} \( \forall i \in V \) \hspace{1cm} (2.3)

\( y_j - x_{ij} \geq 0 \) \hspace{1cm} \( \forall i \in V, \forall j \in C \) \hspace{1cm} (2.4)

\( x_{ij} \in \{0, 1\} \) \hspace{1cm} \( \forall i \in V, \forall j \in C \) \hspace{1cm} (2.5)

\( y_j \in \{0, 1\} \) \hspace{1cm} \( \forall j \in C \) \hspace{1cm} (2.6)

The objective function (2.1) minimizes the total misrepresentation value. Constraint (2.2) ensures that a subset of size \( k \) is selected from the set of candidates. Constraint (2.3) ensures that each voter is assigned to only one candidate in the selected subset. Constraint (2.4) ensures that
each voter is only assigned to a selected candidate. Constraints (2.5), and (2.6) are the integrality constraints.

Let $\mathcal{F}$ denote the set of all feasible solutions of the WDPCC, i.e., the set of all pairs $(\mathbf{x}, \mathbf{y})$ where $\mathbf{x} = [x_{ij}]_{n \times m}$ and $\mathbf{y} = [y_i]_m$ satisfy constraints (2.2) to (2.6). Furthermore, let $f(\mathbf{x}, \mathbf{y})$ denote the objective function value of $(\mathbf{x}, \mathbf{y}) \in \mathcal{F}$. For every fixed $\bar{\mathbf{y}}$ that determines the subset of $k$ candidates from $C$, there are $k^n$ solutions $(\mathbf{x}, \bar{\mathbf{y}})$ in $\mathcal{F}$. Therefore, it is clear that $|\mathcal{F}| = \binom{m}{k} k^n$. In the rest of this chapter, we will interchangeably refer to $\mathbf{y}$ both as a vector of variables $y_j$ and as a corresponding subset of $k$ candidates. The correct meaning will be clear from the context.

2.3 2-Opt Heuristic

In this section, we present a simple 2-Opt heuristic for the winner determination problem under the Chamberlin-Courant system. Let $v : \binom{k}{n} \to \mathbb{N}$ be a function that evaluates the objective function of the best solution with a given $k$-subset of candidates. That is, $v(\bar{\mathbf{y}}) = \min\{f(\mathbf{x}, \bar{\mathbf{y}}) : (\mathbf{x}, \bar{\mathbf{y}}) \in \mathcal{F}\}$. Function $v$ counts the misrepresentation of the candidate (in the input subset) that best represents each voter. Note that, given a $\mathbf{y}$, $v(\mathbf{y})$ can be calculated in linear time (assign each voter to her top-ranked candidate in the $\mathbf{y}$). Algorithm 1 shows the pseudocode of the heuristic.

**Algorithm 1** 2-Opt heuristic for the winner determination problem under the Chamberlin-Courant system

1: **Input:** misrepresentation matrix $\mathbf{r} = [r_{ij}]_{n \times m}$ and number of desired candidates $k \leq m$
2: **Output:** a subset of $k$ candidates
3: Let $\mathbf{y}_{\text{best}}$ be a randomly generated subset of size $k$ of candidates;
4: Set $\text{best}\_\text{solution}\_\text{value} = v(\mathbf{y}_{\text{best}})$;
5: while TRUE do
6: for every pair $(i, j)$ such that $i \in \mathbf{y}_{\text{best}} \land j \in C \setminus \mathbf{y}_{\text{best}}$ do
7: Set $\mathbf{y}' = \mathbf{y}_{\text{best}} \setminus \{i\} \cup \{j\}$;
8: if $v(\mathbf{y}') < \text{best}\_\text{solution}\_\text{value}$ then
9: $\mathbf{y}_{\text{best}} = \mathbf{y}'$
10: $\text{best}\_\text{solution}\_\text{value} = v(\mathbf{y}')$
11: BREAK;
12: if every possible pair $(i, j)$ is checked at step 6, and there was no improvement then
13: BREAK;
14: return $\mathbf{y}_{\text{best}}$;

The heuristic starts with a randomly generated subset of size $k$, $\mathbf{y}_{\text{best}}$, iteratively selects a pair of candidates $i \in \mathbf{y}_{\text{best}}$, $j \in C \setminus \mathbf{y}_{\text{best}}$, and checks if swapping $i$ with $j$ leads to an improved objective function value. The algorithm stops only when no improvement is possible by swapping $i$ with $j$, for every possible pair of candidates $i \in \mathbf{y}_{\text{best}}$, $j \in C \setminus \mathbf{y}_{\text{best}}$. 
2.3.1 Computing the Average Solution Value

Given a set of voters \( V \), a set of candidates \( C \) and a misrepresentation matrix \( r \), let \( A(V, C, r) \) denote the average objective function value among all feasible solutions \( \mathcal{F} \). That is,

\[
A(V, C, r) = \frac{\sum_{(x, y) \in \mathcal{F}} f(x, y)}{|\mathcal{F}|}.
\]

**Theorem 1.** The average objective function value \( A(V, C, r) \) for the WDPCC is equal to

\[
\sum_{i \in V} \sum_{j \in C} r_{ij} \frac{m}{m} = \frac{n(m-1)}{2}.
\]

**Proof.**

\[
Z := \sum_{i \in V} \sum_{j \in C} r_{ij} x_{ij}
\]

\[
\mathbb{E}[Z] = \mathbb{E} \left[ \sum_{i \in V} \sum_{j \in C} r_{ij} x_{ij} \right] = \sum_{i \in V} \sum_{j \in C} r_{ij} \mathbb{E}[x_{ij}]
\]

\[
\mathbb{E}[x_{ij}] = \Pr[x_{ij} = 1] = \Pr[y_j = 1 \text{ and } i \text{ is assigned to } j] = \Pr[y_j = 1] \times \Pr[x_{ij} = 1|y_j = 1]
\]

\[
\Pr[y_j = 1] \text{ is given by }
\]

\[
\Pr[y_j = 1] = \frac{(m-1)}{\binom{m}{k-1}} = \frac{k}{m}
\]

and the conditional probability \( \Pr[x_{ij} = 1|y_j = 1] \) is given by

\[
\Pr[x_{ij} = 1|y_j = 1] = \frac{1}{k}.
\]

Thus, \( \Pr[y_j = 1 \text{ and voter } i \text{ is assigned to } j] \) is given by

\[
\Pr[y_j = 1 \text{ and voter } i \text{ is assigned to } j] = \frac{k}{m} \times \frac{1}{k} = \frac{1}{m}
\]

The average objective function value is given by

\[
A(V, C, r) = \frac{\sum_{i \in V} \sum_{j \in C} r_{ij}}{\sum_{i \in V} |\mathcal{F}|} = \frac{nm(m-1)}{2m} = \frac{n(m-1)}{2}.
\]

Thus, \( \Pr[y_j = 1 \text{ and voter } i \text{ is assigned to } j] \) is given by

\[
\Pr[y_j = 1 \text{ and voter } i \text{ is assigned to } j] = \frac{k}{m} \times \frac{1}{k} = \frac{1}{m}
\]

2.3.2 Computing the Domination Number

In this section we calculate the number of feasible solutions with objective function value no better than the average objective function value \( A(V, C, r) \). We make use of the following result.

**Theorem 2** (Baranyai’s Theorem [10]). If \( k \) divides \( m \), the set of all \( \binom{m}{k} \) \( k \)-subsets of an \( m \)-set may be partitioned into disjoint parallel classes \( A_i \), \( i = 1, 2, \ldots, \binom{m-1}{k-1} \).

Baranyai’s Theorem states that for a set \( S \) and a positive integer \( k \) that divides \( |S| \), the set of all \( k \)-subsets of \( S \) can be split into groups, such that each group is a partition of \( S \).
Lemma 3. For the WDPCC where $k$ divides $m$, there are at least \( \binom{m-1}{k-1} k^{n-1} \) feasible solutions with objective function value greater than or equal to the average objective function value $A(V,C,r)$.

Proof. Recall that for every feasible solution $(x,y) \in \mathcal{F}$, $y$ determines a subset of $k$ candidates, and $x$ assigns a candidate determined by $y$ to each of $n$ voters. In the rest of this Proof, we will identify every such vector $y$ with the subsets of candidates it determines.

Let \( \{\mathcal{Y}_1, \mathcal{Y}_2, \ldots, \mathcal{Y}_{\binom{m-1}{k-1}}\} \) be a partition of all $k$-subsets of candidates, such that each \( \mathcal{Y}_i = \{y_1^i, y_2^i, \ldots, y_{m-i}^i\} \) is a partition of candidates $C$, $i = 1, 2, \ldots, \binom{m-1}{k-1}$. Since $k$ divides $m$, from Theorem 2 it follows that such a partition of all $k$-subsets exists. For example, in the case when $m = 4$ and $k = 2$, such a partition is given by

\[
\{\{\{1, 2\}, \{3, 4\}\}, \{\{1, 3\}, \{2, 4\}\}, \{\{1, 4\}, \{2, 3\}\}\}.
\]

Note that a partition \( \{\mathcal{Y}_1, \mathcal{Y}_2, \ldots, \mathcal{Y}_{\binom{m-1}{k-1}}\} \) also induces a partition \( \{\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_{\binom{m-1}{k-1}}\} \) of \( \mathcal{F} \), where \( \mathcal{F}_i = \{(x, y) \in \mathcal{F} : y \in \mathcal{Y}_i\}, i = 1, 2, \ldots, \binom{m-1}{k-1}\).

Now we fix some $i$, and show how $\mathcal{F}_i$ can be partitioned into small classes of feasible solutions so that the average objective function value in each such class is equal to $A(V,C,r)$.

Before we formally present our construction, we illustrate it with an example: let $n = 2$, $m = 6$, $k = 3$ and \( \mathcal{Y}_i = \{y_1^i = \{1, 2, 3\}, y_2^i = \{4, 5, 6\}\} \). Now consider some solution $s^1$ from $\mathcal{F}_i$. Let $s^1 = (x^1, y^1)$ be such that $y = y_1^i$ and $x_{11}^1 = x_{32}^1 = 1$. We create two additional solutions $s^2 = (x^2, y_1^i)$ and $s^3 = (x^3, y_1^i)$ from $s^1$, by circularly shifting assignments for each voter to the right by one and two candidates, respectively. That is, $x_{12}^2 = x_{23}^2 = 1$ and $x_{13}^3 = x_{21}^3 = 1$. Next we create three other solutions $s^4 = (x^4, y_2^i)$, $s^5 = (x^5, y_2^i)$, $s^6 = (x^6, y_2^i)$ for which the set of candidates will be $y_2^i$. Furthermore, $s^1$, $s^5$, and $s^6$, will be straightforward copies of $s^1, s^2$ and $s^3$, obtained by identifying candidates 1 with 4, 2 with 5, and 3 with 6. That is, $x_{14}^1 = x_{25}^1 = 1$, $x_{15}^5 = x_{26}^5 = 1$ and $x_{16}^6 = x_{24}^6 = 1$. Solutions $s^1, s^2, \ldots, s^6$, are depicted in Figure 2.1. Note that

\[
s^1 \leadsto s^2 \leadsto \cdots \leadsto s^4 \leadsto s^5 \leadsto s^6 \leadsto \cdots
\]

![Diagram](image_url)

Figure 2.1: Example of an equivalence class defined by $\sim_i$, for $\mathcal{Y}_i = \{\{1, 2, 3\}, \{4, 5, 6\}\}$ and $\pi_i$ defined by: $\pi_i(j) = j$ for $j \leq 3$, $\pi_i(j) = j - 3$ for $j > 3$.

For every $s \in V$, $t \in C$, there is exactly one solution $s^j \in \{s^1, \ldots, s^6\}$ for which $x_{st}^j = 1$. Hence
\[ \sum_{j=1}^{6} f(s^j) = \sum_{s \in V} \sum_{t \in C} r_{st}. \] Moreover, the average objective function value among \( s^1, \ldots, s^6 \), is \[ \sum_{j=1}^{6} f(s^j) = \sum_{s \in V} \sum_{t \in C} r_{st} / 6 = A(V, C, r). \] Hence at least one of them is no better than the average. Note that a class of solutions with this property would be obtained if we started with any other starting solution \( s^1 \).

Now we formally present the construction described above. For a fixed \( i \), choose some function \( \pi_i : C \to \{1, 2, \ldots, k\} \) such that \( \{\pi_i(c) : c \in y^j\} = \{1, 2, \ldots, k\} \) for all \( j = 1, 2, \ldots, \frac{m}{k} \). In other words, function \( \pi_i \) gives some arbitrary ordering of elements in \( y^j \), for each \( j = 1, 2, \ldots, \frac{m}{k} \). (In the example of Figure 2.1, \( \pi_i \) was given by \( \pi_i(j) = j \) if \( j \leq 3 \), \( \pi_i(j) = j - 3 \) otherwise.) Now we define a relation \( \sim_i \) on the elements of \( F \) as follows: \( (x', y'_i) \sim_i (x'', y''_i) \) if there exists “shift” \( a \in \{0, 1, \ldots, k-1\} \) such that for all \( j = 1, 2, \ldots, n \), \( x'_{jp} = 1 \) and \( x''_{jq} = 1 \) imply that \( \pi_i(p) \equiv \pi_i(q) + a \pmod{k} \). Note that \( \sim_i \) is an equivalence relation on \( F \). Moreover, if \( S \) is an equivalence class defined by \( \sim_i \), then it has exactly \( m \) elements, and \( \sum_{(x, y) \in S} f(x, y) = \sum_{i \in V} \sum_{j \in C} r_{ij} = mA(V, C, r) \). Hence, there is at least one element of \( S \) with an objective function value no better than \( A(V, C, r) \). Observe that the equivalence classes defined by \( \sim_i \) for all \( i = 1, 2, \ldots, \frac{(m-1)}{k-1} \), provides us a partition of the set \( F \) of all feasible solutions into sets of size \( m \), each of which contains at least one solution with an objective function value no better than the average. Hence, the total number of solutions with objective function values no better than the average is at least

\[ \frac{|F|}{m} = \frac{1}{m} \binom{m}{k} k^n = \frac{1}{m} \frac{m}{k} \binom{m-1}{k-1} k^n = \binom{m-1}{k-1} k^{n-1}, \]

which proves the lemma. \( \square \)

**Remark.** The domination ratio which is the ratio between the number of feasible solution with objective function no better than the 2-opt heuristic and the total number of feasible solution would be \[ \frac{(\frac{(m-1)}{k-1}) k^{n-1}}{\frac{m}{k} k^n} = \frac{k}{k} = \frac{1}{m} \]

Now we use Lemma 3 to get an analogous result for general \( k \), i.e., even when \( k \) does not divide \( m \).

**Theorem 4.** For the WDPCC, there are at least \( \binom{k+1}{k} / k^{-1} k^n \) feasible solutions with an objective function value greater than or equal to the average \( A(V, C, r) \).

**Proof.** Let \( I \) be an instance of the WDPCC where \( k \) does not divide \( m \). Let \( I' \) be the instance obtained from \( I \) by removing \( t = (m \mod{k}) \) candidates with the least sum of misrepresentation value \( \sum_{i \in V} r_{ij} \). Without loss of generality, we assume that \( m - t + 1, m - t + 2, \ldots, m \) are the candidates that are removed. Let \( A_1 \) and \( A_2 \) denote the average objective function values of the solutions for the instance \( I \) and \( I' \), respectively. Then it follows that

\[ A_1 = \frac{\sum_{i \in V} \sum_{j \in C} r_{ij}}{m} = \frac{\sum_{i \in V} \sum_{j=1, \ldots, n-t} r_{ij}}{m - t} = A_2. \] (2.7)
Note that every feasible solution \((x, y)\) for the instance \(I'\) can be transformed to a different feasible solution for the instance \(I\) Namely, just extend \((x, y)\) by setting \(x_{ij} = 0\) and \(y_j = 0\), for all \(j = m - t + 1, m - t + 2, \ldots, m\). From Lemma 3, it follows that for the instance \(I'\), there are at least \(\left(\frac{k!}{k-1} \right) k^{n-1}\) feasible solutions with an objective function value greater than or equal to the average \(A_2\). Then from (2.7), it follows that there are at least \(\left(\frac{k!}{k-1} \right) k^{n-1}\) feasible solutions corresponding to the instance \(I\) with an objective function value no better than the average \(A_1\), which proves the theorem.

Note that Lemma 3 and Theorem 4 hold true also for the more general version of the problem, when the values of \(r\) can be arbitrary, not only when \(\{r_{ij}: j = 1, 2, \ldots, m\} = \{0, 1, \ldots, m - 1\}\) for every \(i \in V\).

### 2.3.3 Domination Analysis of 2-Opt

**Theorem 5.** If a \(k\)-subset \(y = \{j_1, j_2, \ldots, j_k\}\) is locally optimal for 2-Opt, then, \(v(y) \leq A(V, C, r)\).

**Proof.** Let \(y\) be the output of the 2-Opt, and let \(x\) be the corresponding optimal assignment of voters to \(y\), i.e., \(v(y) = f(x, y)\). Then the following inequality is true for any \(V_j\) where \(V_j\) is the set of voters assigned to the candidate \(j\) at the locally optimal solution \((x, y)\).

\[
\sum_{i \in V_j} r_{ij_1} \leq \sum_{i \in V_j} r_{ij_1}
\]

\[
\sum_{i \in V_j} r_{ij_2} \leq \sum_{i \in V_j} r_{ij_2}
\]

\[
\sum_{i \in V_j} r_{ij_3} \leq \sum_{i \in V_j} r_{ij_3}
\]

\[
\vdots
\]

\[
\sum_{i \in V_j} r_{ij_1} \leq \sum_{i \in V_j} r_{ij_m}
\]

Summing the left and right hand side of the above inequalities, we get:

\[
m \sum_{i \in V_j} r_{ij_1} \leq m \sum_{p=1}^{m} \sum_{i \in V_j} r_{ij_p}
\]

Since the above is true for each \(j_q \in y\), the inequalities below follow:

\[
m \sum_{i \in V_j} r_{ij_1} \leq m \sum_{p=1}^{m} \sum_{i \in V_j} r_{ij_p}
\]
\[ m \sum_{i \in V_{j_2}} r_{ij_2} \leq \sum_{p=1}^{m} \sum_{i \in V_{j_2}} r_{ij_p} \]

\[ m \sum_{i \in V_{j_3}} r_{ij_3} \leq \sum_{p=1}^{m} \sum_{i \in V_{j_3}} r_{ij_p} \]

\[ \vdots \]

\[ m \sum_{i \in V_{j_k}} r_{ij_k} \leq \sum_{p=1}^{m} \sum_{i \in V_{j_k}} r_{ij_p} \]

Summing the \( m \) inequalities above, we get:

\[ m \left( \sum_{i \in V_{j_1}} r_{ij_1} + \sum_{i \in V_{j_2}} r_{ij_2} + \cdots + \sum_{i \in V_{j_k}} r_{ij_k} \right) \leq \sum_{j=1}^{m} \sum_{i=1}^{n} r_{ij} \]

Noting that the left hand side is \( m \) times the \( f(x, y) \), we get:

\[ m \cdot f(x, y) = m \cdot v(y) \leq \sum_{j=1}^{m} \sum_{i=1}^{n} r_{ij} \]

By dividing both sides by \( m \), we get:

\[ v(y) \leq \frac{\sum_{j=1}^{m} \sum_{i=1}^{n} r_{ij}}{m} = A(V, C, r) \]

The next theorem shows that the 2-Opt heuristic can obtain a solution with objective function value no worse than the average \( A(V, C, r) \) in polynomial time.

**Theorem 6.** For the winner determination problem under the Chamberlin-Courant system, it takes at most \( \frac{n(m-1)}{2} \) iterations for the 2-Opt heuristic to reach a solution with the objective function value no worse than the average \( A(V, C, r) \).

**Proof.** The worst possible starting solution is when all the voters are assigned to their worst candidates. So the objective function value of the worst solution is \( n(m - 1) \). On the other hand, \( A(V, C, r) = \frac{n(m-1)}{2} \). Since by Theorem 5 we know that 2-Opt reaches \( A(V, C, r) \), and in each iteration the objective function value decreases at least by 1, the maximal number of iterations required to reach \( A(V, C, r) \) is given by

\[ n(m - 1) - \frac{n(m-1)}{2} = \frac{n(m-1)}{2}. \]
2.3.4 Empirical Analysis

We also run different experiments to compare the average cost solution with the solution returned by the 2-Opt heuristics. For this purpose, we run three different types of experiments.

For the Type I experiments, we start by initiating $n_{\text{voters}} = 20$, $n_{\text{candidates}} = 10$, $k = 2$. We generate 60 classes of different instance size. $n_{\text{candidates}}$ and $k$ are fixed for all of the instances, and we only increase $n_{\text{voters}}$ by 10 for each class. We generate 30 different random instances in each class, and then compute the solution returned by 2-Opt heuristics. In the end, we take the average of the 2-Opt solutions of these 30 different instances as a representative of the class to compare against the average cost solution. The results for this comparison are shown in Figure 2.2.

![Figure 2.2: Number of voters is increasing, number of candidates and $k$ are fixed](image)

For the Type II experiments, again we start by initiating $n_{\text{voters}} = 20$, $n_{\text{candidates}} = 10$, $k = 2$, and generate 60 classes of different instance size. Here, $k$ is fixed for all the instances, and we increase $n_{\text{voters}}$ by 10 and $n_{\text{candidates}}$ by 5. The results for this comparison are shown in Figure 2.3.
For the Type III experiments, as before, we start by initiating $n_{\text{voters}} = 20$, $n_{\text{candidates}} = 10$, $k = 2$, and we increase $n_{\text{voters}}$ by 10, $n_{\text{candidates}}$ by 5, and $k$ by 2 in each class. The results for this comparison are shown in Figure 2.4.

As we expected, the solution returned by the 2-Opt heuristics is less than the average cost solution in all experiments.
Chapter 3

Heuristic for Linear Ordering Problem

3.1 Introduction

In the Linear Ordering Problem, we are given a directed graph $G = (V, A)$ with nodes $V = \{1, 2, \ldots, n\}$ and two directed arcs, $(i, j)$ and $(j, i)$, between every pair of nodes $i$ and $j$. Each arc $(i, j) \in A$ has weight $c_{ij}$. Let $< v_1, v_2, \ldots, v_n >$ denote a linear ordering of the nodes $V = \{1, 2, \ldots, n\}$, where $v_1$ precedes $v_2$, $v_2$ precedes $v_3$, and so on, in this ordering. We seek a linear ordering $\sigma$ such that $\sum_{i,j : \sigma(i) < \sigma(j)} c_{ij}$ is maximized, where $\sigma(i) < \sigma(j)$ denotes that $\sigma(i)$ precedes $\sigma(j)$ in the linear ordering $\sigma$. This problem has been shown to be NP-hard [45].

The Linear Ordering Problem (LOP) can be used to model problems in graph theory, such as the feedback arc/node set problem and the node induced acyclic sub digraph problem. The linear ordering problem can also be used to model problems where we need a rank ordering of certain objects using aggregation of individual preferences, so that the ranking we derive matches the individual preferences as closely as possible. This has applications in voting theory, as well as deriving a ranking of players/teams in sports tournaments, to name a few. The linear ordering problem also has applications in machine scheduling, where a set of jobs with precedence constraints need to be scheduled on a single machine. For a detailed treatment of the applications of the linear ordering problem, see [67].

Since it can be used to model many problems of practical importance, a great deal of attention has focused on techniques to obtain either optimal or near optimal solutions to the linear ordering problem. A standard method to solve such a problem optimally is to first obtain an Integer Linear Programming (ILP) formulation for the problem, and then obtain an optimal integer solution, using established methods such as branch and bound. The bounds used in such a branch and bound method may be either a Linear Programming Relaxation bound or a Lagrangian Relaxation bound. However, the computational time required to obtain the optimal solution using such methods grows rapidly with the size of the problem, and these methods become impractical for large-sized problems.
Algorithms and methods to obtain near optimal solutions to these problems have also been developed. An early example of using such an approach with great success was the Local Search Algorithm for the Traveling Salesman Problem [63]. The local search algorithm starts with some initial feasible solution and then iteratively improves the current solution by searching solutions in its neighborhood. Since the quality of the solutions obtained using a local search algorithm often depends on the size of the neighborhoods, techniques to investigate richer neighborhoods have been developed. For a comprehensive survey of such techniques, see [1]. In practice, local search algorithms have been shown to yield the best performance on large instances of computationally hard problems.

More recently, methods have evolved which use a combination of local search methods and exact methods using ILP techniques [39]. In such hybrid methods, an instance of the problem is solved by local search methods, while subproblems are solved optimally, both to explore the neighborhood of a feasible solution, as well as obtain good bounds on the optimal solution. Lourenco first explored hybrid methods [64], where an iterated local search is used to solve the job-shop problem, and ILP techniques are used to optimally solve subproblems to perturb a locally optimal solution. Applegate et al. [9] extend this approach for the Traveling Salesman Problem (TSP) by first running iterated local search multiple times on a TSP instance, and retain the best locally optimal tour each time. Then they construct a restricted graph which has the same set of nodes as the original graph, but with edges that appear at least once in the tours retained. Finally, they solve the TSP optimally on the restricted graph using ILP techniques. to obtain the optimal solution for the original TSP instance.

In a different approach using local search to solve the TSP, Burke et al. [21, 22] employ ILP techniques to optimally solve a subproblem, whose solution corresponds to the optimal solution among a large number of solutions in the neighborhood of a feasible solution. Maniezzo et al. [65] uses an exact method to solve the Quadratic Assignment Problem, with an approximate heuristic method, an approximate non-deterministic tree search in an ant colony optimization algorithm, to obtain a good bound on the optimal solution.

In related work, Mixed Integer Programming (MIP) heuristics use Local Branching (LB), Variable neighborhood Branching (VNB), or Variable neighborhood Decomposition Search (VNDS-MIP), to solve general 0–1 Mixed Integer Linear Programming (MILP) problems. Fischetti et al. [42] use LB, which starts with a feasible solution to the 0–1 MILP $P$. Constraints are then added to $P$ that exclude all solutions with hamming distance greater than or equal to $k$, for some fixed $k$. The problem thus derived is then solved optimally using a MIP solver. Local search is used to find the optimal value for $k$. 

21
3.2 Problem Formulations

The linear ordering problem may be formulated as an integer linear program. For each pair of nodes \( i, j \), \( x_{ij} \) is defined as follows:

\[
x_{ij} = \begin{cases} 
1, & \text{if } i \prec j \text{ (node } i \text{ precedes node } j \text{) in the ordering} \\
0, & \text{otherwise}
\end{cases}
\]

The weight of arc \((i, j)\) in the given instance is denoted \( c_{ij} \). The ILP formulation for LOP is given below [67]. We refer to this as Problem LOP:

Maximize \[ \sum_{(i,j) \in A} c_{ij}x_{ij} \tag{3.1} \]

s.t. \[ x_{ij} + x_{ji} = 1 \quad \forall i, j \in V : i \neq j \tag{3.2} \]
\[ x_{ij} + x_{jk} + x_{ki} \leq 2 \quad \forall i, j, k \in V : i \neq j \neq k \tag{3.3} \]
\[ x_{ij} \in \{0, 1\} \quad \forall i, j \in V \tag{3.4} \]

The objective function (3.1) maximizes the total number of agreements. Constraint (3.2) ensures that either \( i \prec j \) or \( j \prec i \), but not both. Constraint (3.3) prohibits a cycle where \( i \prec j \), \( j \prec k \), and \( k \prec i \). Constraint (3.4) constrains the variables \( x_{ij} \) to take values in the set \( \{0, 1\} \).

The above integer program can be interpreted as follows: either keep the edges with maximum total weight such that the graph is acyclic, or remove the edges with minimum total weight such that the graph is acyclic. With the latter interpretation, we have an alternative formulation for LOP, which is very similar to the integer program for Slater rule introduced in Chapter 1. Here is the alternative formulation:

Minimize \[ \sum_{(i,j) \in A} c_{ji}x_{ij} \tag{3.5} \]

s.t. \[ x_{ij} + x_{ji} = 1 \quad \forall i, j \in V : i \neq j \tag{3.6} \]
\[ x_{ij} + x_{jk} + x_{ki} \geq 1 \quad \forall i, j, k \in V : i \neq j \neq k \tag{3.7} \]
\[ x_{ij} \in \{0, 1\} \quad \forall i, j \in V \tag{3.8} \]

The above formulation ensures that at least one edge is selected and removed from each cycle of length 3 of graph \( G \). It is sufficient to break only cycles of length 3 in order to make the graph acyclic [30].
3.3 Methods Used

We design two heuristics for the linear ordering problem based on a MIP solver. Each heuristic fixes the value (assigns a value of either 0 or 1) for each variable in a subset of the variables, and keeps the remaining variables free. The subproblem that needs to be solved to determine the values of the free variables is then solved optimally using the MIP solver. The only difference between the heuristics is in the method they use to determine the subset of variables whose values are fixed.

3.3.1 LP-Based Heuristic

The first heuristic is based on the LP solution. Algorithm 2 below specifies the steps used to obtain a solution from the optimal LP solution to the LOP. Let \( \hat{x} = \hat{x}_{ij}, (i,j) \in A \) denote the optimal LP solution. If \( n \leq 50 \), a solution \( x^s \) is obtained by solving the LOP optimally, with additional constraints \( x_{ij} = 1 \) if \( \hat{x}_{ij} \geq 0.9 \), and \( x_{ij} = 0 \) if \( \hat{x}_{ij} \leq 0.1 \). The sequence obtained is the solution.

If \( n \geq 50 \), the LOP is solved approximately, with additional constraints \( x_{ij} = 1 \) if \( \hat{x}_{ij} \geq 0.9 \), and \( x_{ij} = 0 \) if \( \hat{x}_{ij} \leq 0.1 \). Instances with size \( n \leq 50 \) are the ones that CPLEX can solve optimally in a reasonable time. The solution obtained from this LP-Based heuristic is then used as a starting solution for the heuristic which will be introduced in the Section 3.3.2. The experiments show that if the number of nodes in the graph become larger than 200, solving the linear program of the LOP becomes very slow, and for a graph with more than 400 nodes, CPLEX fails to return any LP solution due to memory shortage. For these instances, we provide an alternative method based on a randomized greedy algorithm to construct a starting solution. In Chapter 4, we explain how to overcome this problem and show that it is possible to design a combinatorial algorithm to solve the LP of linear ordering problem.

Algorithm 2 LP-Based Heuristic to obtain starting solution

1: Input: Cost Matrix \([C]_{n \times n}\)
2: Output: An ordering \( v = < v_1, v_2, \ldots, v_n > \) of the nodes in the graph \( G = (V, A) \)
3: Solve LP relaxation of the problem and get the \( \hat{x} = \{ x_{ij} \}_{i,j \in N} \);
4: \( \forall i,j \), set the variable \( x_{ij} = 1 \) where \( \hat{x}_{ij} \geq 0.9 \);
5: \( \forall i,j \), set the variable \( x_{ij} = 0 \) where \( \hat{x}_{ij} \leq 0.1 \);
6: Solve LOP optimally with the additional constraints 2 and 3 above;
7: if \( n \leq 50 \) then return optimal ordering \( v^s \);
8: if \( n \geq 50 \) then
9: Solve LOP approximately with MIP solver;
10: Obtain the intermediate ordering \( v^i = < v^i_1, v^i_2, \ldots, v^i_n > \)
11: Generate two subproblems P1 with nodes \( \{ v^i_1, v^i_2, \ldots, v^i_k \} \) and P2 with nodes \( \{ v^i_{k+1}, v^i_{k+2}, \ldots, v^i_n \} \) (where \( k = \lfloor n/2 \rfloor \));
12: Solve each subproblem optimally by CPLEX and get subsequences \( v^1 = < v^1_1, \ldots, v^1_k >, v^2 = < v^2_{k+1}, \ldots, v^2_n > \);
13: Concatenate the two subsequences to return sequence \( v^s = < v^s_1, \ldots, v^s_k, v^s_{k+1}, \ldots, v^s_n > \);
3.3.2 Mixed Integer Program Heuristic

Let $v^i = <v^i_1, v^i_2, \ldots, v^i_n>$ denote the intermediate ordering provided by LP-Based heuristic (or any other heuristic). Let $k = \lceil n/2 \rceil$, we form two subproblems, P1 with nodes $\{v^1_k, \ldots, v^1_n\}$, and P2 with nodes $\{v^2_k, v^2_{k+2}, \ldots, v^2_n\}$. We solve subproblems P1 and P2 optimally using CPLEX, to obtain subsequences $v^1 =<v^1_1, v^1_2, \ldots, v^1_k>$ and $v^2 =<v^2_k, v^2_{k+2}, \ldots, v^2_n>$. The two subsequences are then concatenated to obtain the starting sequence $v^s =<v^1_1, v^1_2, \ldots, v^2_k, v^2_{k+1}, \ldots, v^2_n>$. Algorithm 4 below outlines the MIP heuristic we use to solve the LOP. We first derive a feasible integer solution to the problem from an optimal LP solution to LOP. We call such a solution a starting solution.

Any feasible integer solution, including the starting solution, specifies an ordering of the nodes. Let the starting solution, denoted $v^s$, specify the ordering $<v^s_1, v^s_2, \ldots, v^s_n>$. Given a starting solution $v^s$ to the LOP, and a set $N$ of neighborhood solutions feasible to the LOP, we obtain the optimal solution in set $N$ using the MIP solver CPLEX (Algorithm 3). We use the solution thus obtained as our new starting solution, and repeat this procedure for a number of iterations ($I$). We return the best feasible solution obtained.

We describe below the set $N$ of neighboring solutions to feasible solution $v =<v_1, v_2, \ldots, v_n>$. The solution $v$ can be also considered as a sequence of the integers $v_1, v_2, \ldots, v_n$, where $<v_1, v_2, \ldots, v_n>$ is a permutation of the integers $<1, 2, \ldots, n>$. Given such a sequence, we can define a subsequence $v_{i:j} =<v_i, v_{i+1}, \ldots, v_j>$, where $1 \leq i \leq j \leq n$. The neighborhood $N_{ij}$ (with respect to the subsequence $v_{i:j}$) is defined as the set of all orderings $w=<w_1, w_2, \ldots, w_n>$, where $w_i = v_i$ for $1 \leq i \leq j$, and $w_j = v_i$ for $j+1 \leq i \leq n$. Note that in the ordering $w$, the subsequence $w_{1:i-1} =<w_1, w_2, \ldots, w_{i-1}>$ is identical to the subsequence $v_{1:i-1} =<v_1, v_2, \ldots, v_{i-1}>$, and the subsequence $w_{j+1:n} =<w_{j+1}, w_{j+2}, \ldots, w_n>$ is identical to the subsequence $v_{j+1:n} =<v_{j+1}, v_{j+2}, \ldots, v_n>$. However, the subsequence $w_{i:j}=<w_i, w_{i+1}, \ldots, w_j>$ can be any reordering of the subsequence $v_{i:j} =<v_i, v_{i+1}, \ldots, v_j>$. In addition, for every pair $k, l, 1 \leq k \leq i-1, i+1 \leq l \leq j$, $w_k < w_l$, and for every pair $p, q, i \leq p \leq j, j+1 \leq q \leq n$, $w_p < w_q$. In other words, every node in the subsequence $w_{1:i-1}$ precedes every node in the subsequence $w_{i:j}$, and every node in the subsequence $w_{i:j}$ precedes every node in the subsequence $w_{j+1:n}$. Thus the size of the neighborhood set $N_{ij}$ is $(j-i+1)!$, which is exponential in the size of the subsequence $s = (j-i+1)$. The subsequence itself is specified in terms of the start position $i$, and the size of the subsequence $s$. 

24
**Algorithm 3** MIP Solver to obtain optimal solution in neighborhood

1: **Input:** An ordering $v = <v_1, v_2, \ldots, v_n>$, and a subsequence $v_{i,j}$

2: **Output:** An ordering $\hat{v} = <\hat{v}_1, \hat{v}_2, \ldots, \hat{v}_n>$ of the nodes in the graph $G = (V, A)$

3: Solve LP relaxation of the problem and get the $\hat{x} = \{x_{ij}\}_{i,j \in N}$;

4: for all $s, t \in [1, 2, \ldots, n]$ do

5: \hspace{1em} if $s, t \not\in [i, \ldots, j]$ and $v_s \succ v_t$ then

6: \hspace{2em} $x_{s,t} = 1$;

7: \hspace{2em} $x_{t,s} = 0$;

8: \hspace{1em} else if $s \not\in [i, \ldots, j]$ and $t \in [i, \ldots, j]$ and $v_s \succ v_t$ then

9: \hspace{2em} $x_{s,t} = 1$;

10: \hspace{2em} $x_{t,s} = 0$;

11: \hspace{1em} else if $s \not\in [i, \ldots, j]$ and $t \in [i, \ldots, j]$ and $v_s < v_t$ then

12: \hspace{2em} $x_{s,t} = 0$;

13: \hspace{2em} $x_{t,s} = 1$;

14: Solve LOP with the additional constraints specified above; return the ordering obtained $\hat{v}$;

**Algorithm 4** Mixed Integer Program Heuristic

1: **Input:** Cost Matrix $[C]_{n \times n}$

2: **Output:** An ordering $\hat{v} = <v_1, v_2, \ldots, v_n>$ of the nodes in the graph $G = (V, A)$

3: Run Algorithm 2 (LP-Based Heuristic) and get starting solution $v^*$;

4: Let $|N|$ denote the size of the neighborhood set;

5: $N = 5$ (Starting size for threshold is 5);

6: while number of iterations $i < I$ do

7: \hspace{1em} Pick a subsequence $v^*_{i,j}$ of size $N$ randomly from $\hat{v}$;

8: \hspace{1em} Run Algorithm 3 with the inputs: ordering $\hat{v}$, subsequence $\hat{v}_{i,j}$;

9: \hspace{1em} Obtain output ordering $\hat{v}$;

10: \hspace{1em} Update $v^* = \hat{v}$;

11: \hspace{1em} if there is no improvement after a number of iterations then

12: \hspace{2em} $N = N + \delta$ ($\delta$ is 5 for earlier iteration and is 2 for later iterations)

3.3.3 Alternative Algorithm for Starting Solution

The LP-Based heuristic provides the starting solution in Step 1 of Algorithm 4. When the size of the graph is large (more 200 nodes), we use a simple local search algorithm called Greedy Randomized Adaptive Search Procedures (GRASP) to provide the starting solution. This procedure was first used by Feo et al. [41] to solve computationally difficult set covering problems. Marti et al. [67] used GRASP for linear ordering problem. In each iteration of GRASP, a random greedy solution is
constructed and then will be improved by a local search method. Then, there are two phases within each GRASP iteration: construction and improvement phase. Algorithm 5 is the pseudocode of GRASP.

**Algorithm 5** GRASP

1: \( i = 1 \);
2: while number of iterations \( i < I \) do
3: Construct a solution \( x_i \) (Construction phase);
4: Obtain the solution \( x'_i \) by applying a local search algorithm to improve \( x_i \) (Improvement phase);
5: if \( x'_i \) is a better solution than \( x_i \) then
6: Update the best solution found so far;
7: \( i = i + 1 \);

**Construction Phase**

In the construction phase, an initial solution is built iteratively by considering one node at a time. Each node is added by using a greedy function as a guide. The node chosen at any iteration of the construction phase is a function of those chosen previously. Thus relevant information is updated from one construction step to the next one. Specifically, at each iteration of the construction phase, GRASP maintains a list of nodes which can be added to the partial solution under construction to obtain a feasible, complete solution. All candidate nodes are evaluated according to a greedy function (evaluator) to select the next node to be added in the construction.

Marti et al. [67] used three different greedy evaluators. The first evaluator, denoted \( e_1(i) \) of node \( i \), is the sum of the elements of the \( i^{th} \) row of the cost matrix \( C \):

\[
e_1(i) = \sum_{j=1}^{n} c_{ij}
\]

The second evaluator, denoted \( e_2(i) \) of node \( i \), is the difference between \( \text{colmax} \) and the sum of the elements in the \( i^{th} \) column of the cost matrix, where \( \text{colmax} \) is the maximum of the sum of columns:

\[
e_2(i) = \text{colmax} - \sum_{j=1}^{n} c_{ji}
\]

Finally, the third evaluator, denoted \( e_3(i) \) of node \( i \), is the ratio between the sum of the elements of the \( i^{th} \) row and sum of the elements of the \( i^{th} \) column of the cost matrix:

\[
e_3(i) = \frac{\sum_{j=1}^{n} c_{ij}}{\sum_{j=1}^{n} c_{ji}}
\]
A complete ordering (permutation) of the $n$ nodes of the graph is obtained in the construction phase. Assume that list $N$ contains all the nodes of the graph. At the beginning of the construction phase, a restricted list of candidate nodes, $RL$, is created. Each node is evaluated with greedy evaluators ($e_1, e_2$ or $e_3$), and the best ones are picked and added to $RL$ (RL has a limited size). At each iteration, one node is chosen randomly from $RL$ and added to the partial order. Then the list $RL$ is updated by adding some of the best nodes from $N$ (as per the greedy evaluator). After $n$ iterations, we obtain a complete ordering.

**Improvement Phase**

The local search that Marti et al. [67] use for their GRASP is presented and developed by Laguna et al. [61] and is called insertion. Let us assume that we have a feasible solution $v = <v_1, v_2, \ldots, v_n>$. Laguna et al. define $move(v_j, i)$ as deleting $v_j$ from its current position $j$ in the current feasible solution $v$, and inserting it at position $i$ (i.e., between $v_{i-1}$ and $v_i$). The GRASP algorithm improves the solution generated in the construction phase iteratively as long as the objective function increases. At each step, it considers a node $v_j$ in the ordering $v$, and examine all positions $i$ from 1 to $n$, in search for the position $i^*$ with the best $move(v_j, i^*)$. If $move(v_j, i^*)$ improves the objective function, it will be performed, otherwise it will be discarded and the local search will test the next node $v_{j+1}$.

We run several experiments to compare the quality of the solutions returned by Algorithm 4 when using GRASP versus LP-Based heuristic to provide the starting solution.

### 3.4 Empirical Analysis

The computational experiments were conducted on an Intel Core i7 with 2.8 GHz 64-bit processor, 16.0 GB of RAM and OSX El Captain 64-bit as the Operating System. The heuristics were implemented in C++. The LP and MILP problems were solved with CPLEX 12.5. All the experiments are run with a time limitation of 500 seconds.

#### 3.4.1 Data Set

For our experiments, we use the data sets collected by Marti et al. [68].

**Instances RandAI**: These instances are generated from a uniform distribution in the range $[0, 100]$. They were initially generated from a uniform distribution in the range $[0, 25000]$ and then sampled from the much narrower range $[0, 100]$. These are the harder instances. The number of nodes in these instances are 100, 150, and 200.

**Instances RandAII**: These contain easier instances which are generated from randomly generated permutations. Again, the number of nodes in these instances are 150 and 200.

**Instances of XLOLIB**: These benchmark instances have been created and used by Schiavinotto [80]. These instances were generated from the real-world tables by replicating them to obtain larger
instances of RandB: For these random instances, the superdiagonal entries of cost matrix \( C \) are drawn uniformly distributed from the interval \([0, U_1]\) and the subdiagonal entries from \([0, U_2]\), where \( U_1 \geq U_2 \).

Special Instances: These are further problem instances that were used for experiments in some publications. The atp instances were created from the results of ATP tennis tournaments in 1993/1994. For a detailed treatment of the methods that have been used to solve the Linear Ordering Problem, and a description of these instances, see [68].

3.4.2 Computational Results

We run three different types of experiments to show the performance of MIP heuristic. The first experiment compares the performance of the MIP heuristic with different starting solutions. The second experiment compares the performance of MIP heuristic versus solving the entire problem by CPLEX. And finally, we compare the MIP heuristic results with the best-known heuristic in the literature.

GRASP versus LP-Based Heuristic

We consider the following 4 algorithms to construct a starting solution for MIP heuristic: LP-Based heuristic, GRASP with greedy function \( e_1 \) (G1), GRASP with greedy function \( e_2 \) (G2), GRASP with greedy function \( e_3 \) (G3). In this experiment, we use the instance N-t1d100.01 from RandA1 data set. We have generated a set of 100 solutions with each of 4 generation methods. Figure 3.1 shows in a box-plot representation, the value of 100 solutions generated with each method. Since the linear ordering problem is a maximization problem, the LP-Based heuristic method obtains the best results. G2 solutions are more diverse than the G1, and G3.
Both CPLEX and the MIP heuristic have the same time limit (500 seconds) for these experiments. Solving LP for graphs with more than 200 nodes is time-consuming. For these instances we use the solution returned by G3 as a starting solution. We summarize the computational results in Table 3.1, Table 3.2, and Figure 3.2. Table 3.1 and Figure 3.2a show experiments on instances of RandAI, while Table 3.2 and Figure 3.2b show experiments on instances of RandAII. Table 3.1 shows that for the harder instances for the sizes 100 and 150, the largest integrality gap obtained by CPLEX is almost 24% while the largest integrality gap obtained by the heuristic is 15%. For the easier instances, the largest integrality gap obtained by the CPLEX is 2.14% and the largest integrality gap obtained by the heuristic is 0.85%.
Table 3.1: Computational results for instances RandAI

<table>
<thead>
<tr>
<th>Instance</th>
<th>CPLEX</th>
<th>MIP Heuristic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sol</td>
<td>Gap %</td>
</tr>
<tr>
<td>N-t1d100.01</td>
<td>96637</td>
<td>18.45%</td>
</tr>
<tr>
<td>N-t1d100.02</td>
<td>97508</td>
<td>16.99%</td>
</tr>
<tr>
<td>N-t1d100.03</td>
<td>99330</td>
<td>18.64%</td>
</tr>
<tr>
<td>N-t1d100.04</td>
<td>99457</td>
<td>18.28%</td>
</tr>
<tr>
<td>N-t1d100.05</td>
<td>99113</td>
<td>18.59%</td>
</tr>
<tr>
<td>N-t1d100.06</td>
<td>96924</td>
<td>20.77%</td>
</tr>
<tr>
<td>N-t1d100.07</td>
<td>97703</td>
<td>19.06%</td>
</tr>
<tr>
<td>N-t1d100.08</td>
<td>97224</td>
<td>19.06%</td>
</tr>
<tr>
<td>N-t1d150.01</td>
<td>213538</td>
<td>20.97%</td>
</tr>
<tr>
<td>N-t1d150.02</td>
<td>213433</td>
<td>21.37%</td>
</tr>
<tr>
<td>N-t1d150.03</td>
<td>211917</td>
<td>23.66%</td>
</tr>
<tr>
<td>N-t1d150.04</td>
<td>212812</td>
<td>21.99%</td>
</tr>
<tr>
<td>N-t1d150.05</td>
<td>214151</td>
<td>21.34%</td>
</tr>
<tr>
<td>N-t1d150.06</td>
<td>212907</td>
<td>22.13%</td>
</tr>
<tr>
<td>N-t1d150.07</td>
<td>214337</td>
<td>21.73%</td>
</tr>
<tr>
<td>N-t1d150.08</td>
<td>216617</td>
<td>20.46%</td>
</tr>
</tbody>
</table>
Table 3.2: Computational results for instances RandAll

<table>
<thead>
<tr>
<th>Instance</th>
<th>CPLEX</th>
<th>MIP Heuristic</th>
<th>CPLEX</th>
<th>MIP Heuristic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sol</td>
<td>Gap%</td>
<td>Best Bound</td>
<td>Sol</td>
</tr>
<tr>
<td>N-t2d150.01</td>
<td>75296</td>
<td>1.3%</td>
<td>76276</td>
<td>76021</td>
</tr>
<tr>
<td>N-t2d150.02</td>
<td>72889</td>
<td>1.26%</td>
<td>73811</td>
<td>73617</td>
</tr>
<tr>
<td>N-t2d150.03</td>
<td>68926</td>
<td>1.4%</td>
<td>69894</td>
<td>69703</td>
</tr>
<tr>
<td>N-t2d150.04</td>
<td>73189</td>
<td>1.29%</td>
<td>74136</td>
<td>73960</td>
</tr>
<tr>
<td>N-t2d150.05</td>
<td>78890</td>
<td>1.21%</td>
<td>79847</td>
<td>79723</td>
</tr>
<tr>
<td>N-t2d150.06</td>
<td>74684</td>
<td>1.28%</td>
<td>75604</td>
<td>75438</td>
</tr>
<tr>
<td>N-t2d150.07</td>
<td>73143</td>
<td>1.26%</td>
<td>74067</td>
<td>73852</td>
</tr>
<tr>
<td>N-t2d150.08</td>
<td>66586</td>
<td>1.82%</td>
<td>67803</td>
<td>67463</td>
</tr>
<tr>
<td>N-t2d150.09</td>
<td>146132</td>
<td>1.48%</td>
<td>148295</td>
<td>147704</td>
</tr>
<tr>
<td>N-t2d200.01</td>
<td>142566</td>
<td>1.63%</td>
<td>144904</td>
<td>144100</td>
</tr>
<tr>
<td>N-t2d200.02</td>
<td>139632</td>
<td>1.77%</td>
<td>142105</td>
<td>141260</td>
</tr>
<tr>
<td>N-t2d200.03</td>
<td>149170</td>
<td>1.54%</td>
<td>151471</td>
<td>149520</td>
</tr>
<tr>
<td>N-t2d200.04</td>
<td>148694</td>
<td>1.45%</td>
<td>150854</td>
<td>150176</td>
</tr>
<tr>
<td>N-t2d200.05</td>
<td>139592</td>
<td>1.73%</td>
<td>142009</td>
<td>141100</td>
</tr>
<tr>
<td>N-t2d200.06</td>
<td>148132</td>
<td>1.51%</td>
<td>150379</td>
<td>149126</td>
</tr>
<tr>
<td>N-t2d200.07</td>
<td>148042</td>
<td>1.6%</td>
<td>150415</td>
<td>149650</td>
</tr>
<tr>
<td>N-t2d200.08</td>
<td>140190</td>
<td>1.82%</td>
<td>142747</td>
<td>141896</td>
</tr>
<tr>
<td>N-t2d200.09</td>
<td>147990</td>
<td>1.51%</td>
<td>150232</td>
<td>149612</td>
</tr>
<tr>
<td>N-t2d200.10</td>
<td>145990</td>
<td>1.55%</td>
<td>148262</td>
<td>147398</td>
</tr>
<tr>
<td>N-t2d200.11</td>
<td>150794</td>
<td>1.5%</td>
<td>153064</td>
<td>152394</td>
</tr>
<tr>
<td>N-t2d200.12</td>
<td>135922</td>
<td>1.90%</td>
<td>138514</td>
<td>137572</td>
</tr>
<tr>
<td>N-t2d200.13</td>
<td>142542</td>
<td>1.78%</td>
<td>145083</td>
<td>144108</td>
</tr>
<tr>
<td>N-t2d200.14</td>
<td>145680</td>
<td>1.67%</td>
<td>148114</td>
<td>147356</td>
</tr>
<tr>
<td>N-t2d200.15</td>
<td>129950</td>
<td>2.13%</td>
<td>132728</td>
<td>131850</td>
</tr>
<tr>
<td>N-t2d200.16</td>
<td>149908</td>
<td>1.17%</td>
<td>151663</td>
<td>150948</td>
</tr>
<tr>
<td>N-t2d200.17</td>
<td>35572</td>
<td>1.925%</td>
<td>38182</td>
<td>37272</td>
</tr>
<tr>
<td>N-t2d200.18</td>
<td>145180</td>
<td>1.28%</td>
<td>147051</td>
<td>146448</td>
</tr>
<tr>
<td>N-t2d200.19</td>
<td>141700</td>
<td>1.77%</td>
<td>144221</td>
<td>143428</td>
</tr>
<tr>
<td>N-t2d200.20</td>
<td>145238</td>
<td>1.57%</td>
<td>147524</td>
<td>146828</td>
</tr>
<tr>
<td>N-t2d200.21</td>
<td>143480</td>
<td>1.56%</td>
<td>145731</td>
<td>144932</td>
</tr>
<tr>
<td>N-t2d200.22</td>
<td>149658</td>
<td>1.41%</td>
<td>151769</td>
<td>151208</td>
</tr>
<tr>
<td>N-t2d200.23</td>
<td>147476</td>
<td>1.52%</td>
<td>149723</td>
<td>149078</td>
</tr>
</tbody>
</table>
MIP Heuristic versus Other Heuristics

The MIP heuristic cannot beat the best-known heuristics on the tested data sets. Marti et al. [68] experiments show that memetic algorithm obtained the best-known results. The MIP heuristic can only achieve comparable results on instances of data set RandB, and for all other data sets, it is behind the best-known solution.
Chapter 4

Combinatorial Algorithm for Solving Linear Program of Linear ordering Problem

4.1 Introduction

In Chapter 3, we designed a heuristic for the linear ordering problem. The heuristic starts with an initial solution provided by rounding the LP solution to an integer solution. It then performs local improvement on the initial solution. The experimental results showed that for instances larger than 200 nodes, the LP solver for the LOP was very slow. Therefore using the LP solver to provide an initial solution for the heuristic is impractical. In this chapter, we focus on solving the linear program of the linear ordering problem efficiently.

In practice, the LP solver uses the Simplex algorithm (or a variant) to solve linear programs. Dantzig et al. [34] introduced the primal-dual method, which is a general algorithm for solving linear programs. This method was first used to solve the linear program for some network problems. Dantzig et al. [34], Dijkstra [38], and Ford and Fulkerson [43] showed that it could be used to design a combinatorial algorithm for many graph related problems. In this chapter, we will show that the LP of the linear ordering problem can be solved using the primal-dual algorithm. We start with the alternative formulation for the linear ordering problem we introduced in Chapter 3. We then write
the dual, the restricted primal and the dual of the restricted primal for this formulation.

\[
\begin{align*}
\text{Minimize} & \quad \sum_{(i,j) \in A} c_{ji}x_{ij} \\
\text{s.t.} & \quad x_{ij} + x_{ji} = 1 \quad \forall i, j \in V \\
& \quad x_{ij} + x_{jk} + x_{ki} \geq 1 \quad \forall i, j, k \in V \\
& \quad x_{ij} \in \{0, 1\} \quad \forall i, j \in V
\end{align*}
\]

(4.1) \hspace{2cm} (4.2) \hspace{2cm} (4.3) \hspace{2cm} (4.4)

First, we briefly explain the primal-dual framework.

4.2 Primal-Dual Algorithm

Before we outline the primal-dual framework, we recall the complementary slackness conditions for the linear programming problem. Suppose that we have a linear program (P) in standard form:

\[
\begin{align*}
\text{min} & \quad c^T x \\
\text{s.t.} & \quad Ax = b \\
& \quad x \geq 0
\end{align*}
\]

Its dual (D) is given below:

\[
\begin{align*}
\text{max} & \quad y^T b \\
& \quad y^T A \leq c \\
& \quad y \leq 0
\end{align*}
\]

Then the complementary slackness conditions for the primal-dual pair above are specified in the following theorem. We refer to \( j^{th} \) column of \( A \) as \( A_j \) and \( i^{th} \) row of \( A \) as \( a_i^T \).

**Theorem 7. (Complementary slackness conditions)**

A pair \((x, y)\) respectively feasible in a primal-dual pair is optimal if and only if

\[
\begin{align*}
y_i(a_i^T x - b_i) = 0 & \quad \forall i \\
x_j(c_j - y^T A_j) = 0 & \quad \forall j
\end{align*}
\]

We now outline the primal-dual method. Consider a solution \( y \) which is feasible to (D). Then some of the inequalities \( y^T A_j \leq c_j \) are satisfied with strict inequality while the rest are satisfied as equalities. Let \( J \) be the set of indices for which \( y^T A_j = c_j \). (The inequalities corresponding
to set \( J \) are satisfied as equalities.) According to the complementary slackness Theorem, for any feasible solution \( x \) to be optimal, \( x_j = 0 \) for \( j \notin J \) (\( x_j \) unrestricted for \( j \in J \)). The primal-dual method tries to work only with columns/variables in \( J \), called admissible columns, to construct a feasible solution \( x \) for the primal. To do this, it creates a Restricted Primal (RP) using only the columns/variables in set \( J \). The RP for the (D) is given below:

\[
\begin{align*}
\min \quad & \sum_{i=1}^{m} s_i \\
\text{s.t.} \quad & \sum_{j \in J} a_{ij} x_j + s_i = b_i & \quad i = 1, 2, \ldots, m \\
& x_j = 0 & \quad j \notin J \\
& x_j \geq 0 & \quad j \in J \\
& s_i \geq 0 & \quad i = 1, 2, \ldots, m
\end{align*}
\]

An auxiliary variable \( s_i \) is introduced for each constraint \( i, i = 1, 2, \ldots, m \). If the objective function value of the optimal solution is 0, then this solution is also feasible (as well as optimal) for (P) and we are done (each \( s_i \) is 0, and \( \sum_{j \in J} a_{ij} x_j = b_i \) for \( i = 1, 2, \ldots, m \)). If the objective function value of the optimal solution to (RP) is not 0, then it means \( \sum_{j \in J} a_{ij} x_j \neq b_i \) for some \( i = 1, 2, \ldots, m \). This implies that solution \( x \) is not feasible in (P), and we have to search for another solution. To search for another solution \( x \), the primal-dual algorithm works with the dual of (RP), called (DRP). It finds the optimal solution \( \bar{y} \) to (DRP) and uses this to modify the feasible solution \( y \) to (D). The (DRP) is given below:

\[
\begin{align*}
\max \quad & y^T b \\
\text{s.t.} \quad & y^T A_j \leq 0 & \quad j \in J \\
& y_i \leq 1 & \quad i = 1, 2, \ldots, m \\
& y_i \leq 0
\end{align*}
\]

The new feasible dual solution \( y^* \) is given by

\[
y^* = y + \theta \bar{y}
\]

The primal-dual algorithm chooses \( \theta \) so that \( y^* \) stays feasible in \( D \) and also increases the dual objective function value. For a more comprehensive explanation of the primal-dual algorithm and the choice of \( \theta \), we refer to Chapter 5 in [73].
We now apply the primal-dual method to solve the LP formulation of the LOP. The inequality constraints (4.3) are rewritten as equality constraints below by introducing slack variables $s_{ijk} \geq 0$ for each triple $i, j, k \in V$. The primal formulation (P) in the standard form is as follows:

Minimize \[
\sum_{(i,j) \in A} c_{ji}x_{ij}
\]

s.t. \[
x_{ij} + x_{ji} = 1 \quad \forall i, j \in V : i < j \\
x_{ij} + x_{jk} + x_{ki} - s_{ijk} = 1 \quad \forall i, j, k \in V \\
x_{ij} \geq 0 \quad \forall i, j \in V \\
s_{ijk} \geq 0 \quad \forall i, j, k \in V
\]

The corresponding dual formulation (D) is given below:

Maximize \[
\sum_{(i,j) \in A} y_{ij} + \sum_{i,j,k \in V} z_{ijk}
\]

s.t. \[
y_{ij} + \sum_{k \in V} z_{ijk} \leq c_{ji} \quad \forall i, j \in V \\
z_{ijk} \geq 0 \quad \forall i, j, k \in V \\
y_{ij} \leq 0 \quad \forall i, j \in V \\
z_{ijk} \leq 0 \quad \forall i, j, k \in V
\]

Given any feasible solution $y, z$ to (D), set $J$ contains every pair of nodes $(i, j) \in V$ for which the constraint $y_{ij} + \sum_{k \in V} z_{ijk} = c_{ji}$. Set $K$ contains every triple $(i, j, k) \in V$ for which the constraint
Using the sets \( J \) and \( K \), we obtain the restricted primal (RP) formulation below:

Minimize \[
\sum_{(i,j) \in A} a_{ij} + \sum_{i,j,k \in V} b_{ijk}
\] (4.15)

s.t. \[
\begin{align*}
    x_{ij} + x_{ji} + a_{ij} &= 1 & \forall i, j \in V \\
    x_{ij} + x_{jk} + x_{ki} - s_{ijk} + b_{ijk} &= 1 & \forall i, j, k \in V \\
    s_{ijk} &= 0 & \forall (i, j, k) \notin K \\
    x_{ij} &\geq 0 & \forall (i, j) \in J \\
    s_{ijk} &\geq 0 & \forall (i, j, k) \in K \\
    b_{ijk} &\geq 0 & \forall i, j, k \in V \\
    a_{ij} &\geq 0 & \forall i, j \in V
\end{align*}
\] (4.16 - 4.23)

Finally, the formulation for the dual of the restricted primal (DRP) is:

Maximize \[
\sum_{(i,j) \in A} y_{ij} + \sum_{i,j,k \in V} z_{ijk}
\] (4.24)

s.t. \[
\begin{align*}
    y_{ij} + \sum_{k \in V} z_{ijk} &\leq 0 & \forall (i,j) \in J \\
    z_{ijk} &\geq 0 & \forall (i, j, k) \notin K \\
    z_{ijk} &\leq 1 & \forall i, j, k \in V \\
    y_{ij} &\leq 1 & \forall i, j \in V \\
    y_{ij} &\leq 0 & \forall i, j \in V \\
    z_{ijk} &\leq 0 & \forall i, j, k \in V
\end{align*}
\] (4.25 - 4.30)

We will show that the (DRP) can be solved combinatorially. The algorithm can be considered as a further instance of the concept of augmentation familiar to us from the solution of the maximum matching and the maximum flow problem [14, 73]. The method and proofs in this chapter can be seen as a generalization of the work on the maximum charge problem by Krishnamurti et al. [60]

### 4.3 Graphical Representation of DRP

Given the DRP formulation, we can construct a graph \( G = (V, E) \) such that

\[
V = \{y_{ij}| i, j \in J\} \cup \{z_{ijk}| z_{ijk} \in L\}
\]
where set $L$ contains $z_{ijk}$ if at least one of $(i, j)$, $(j, k)$, or $(k, i)$ appear in set $J$. For every $y_{ij} \in J$, there is an edge between $y_{ij}$ and $z_{ijk}$ for any $k$ ($k \neq i \neq j$). Given below is an example of this graph:

![Figure 4.1: Example of DRP graph](image)

Since the DRP is a maximization problem, each $y_{ij}$ and $z_{ijk}$ that does not appear in the DRP graph may be set to 1. The following properties of the DRP graph can be easily derived from the primal, dual, RP, and DRP formulations:

- leaf nodes can only be $z$ nodes
- the degree of each $z$ node is at most 3
- the degree of each $y$ node is at most $|V| - 2$
- the upper bound on the capacity of each node ($z$ or $y$) is at most 1
- there is no lower bound on the capacity of nodes, except for the $z$ nodes that are in set $K$. The lower bound for these nodes is 0
- all the $y$ nodes in this graph are in set $J$
- $z_{ijk}$ appears in the graph if and only if at least one of the pairs $(i, j)$, $(j, k)$, or $(k, i)$ appear in set $J$
- $\forall i, j$, $y_{ij} = y_{ji}$
- for any $i, j, k$, only two $z$ nodes appear in the DRP graph, one in decreasing order of this triple, and the other in increasing order

### 4.3.1 Definitions

**Definition 4.3.1** $(A(y, z))$. A feasible solution $A$ is a function $a : V \rightarrow R$ defined on $V$ such that:

(i) $-\infty < a(y) \leq 1 \forall y \in V$

(ii) $-\infty < a(z) \leq 1 \forall z \in V \setminus K$
(iii) \(0 < a(z) \leq 1 \ \forall z \in V \cap K\)

**Definition 4.3.2** \((N(z_i, N(y_j)).\)

\[
N(z_i) = \{y | (z_i, y) \in E\}
\]

\[
N(y_j) = \{z | (y_j, z) \in E\}
\]

**Definition 4.3.3** (Plus and Minus Node). A \(z\) or \(y\) node is called a plus node with respect to optimal solution \(A^*(y, z)\) and the current feasible solution \(A(y, z)\), if \(a^*(z) > a(z)\) for a \(z\) node, and \(a^*(y) > a(y)\) for a \(y\) node.

Conversely, a \(z\) or \(y\) node is called a minus node with respect to optimal solution \(A^*(y, z)\) and the current feasible solution \(A(y, z)\), if \(a^*(z) < a(z)\) for a \(z\) node, and \(a^*(y) < a(y)\) for a \(y\) node.

**Definition 4.3.4** (Walk). A vertex-edge sequence (also called a walk) \(S\) of length \(l\) is a sequence \(S = \{z_1, e_1, y_1, e_2, z_2, e_3, y_3, \ldots, y_{l-1}, e_l, z_l\}\) where edge \(e\) either connects vertex \(z\) to vertex \(y\) or vertex \(y\) to vertex \(z\); note that in \(S\) the same vertex and the same edge may occur, with different subscripts, more than once.

**Definition 4.3.5** (Path). \(S\) is a path \(P\) if and only if the vertices \(z_1, y_1, z_2, \ldots, y_{l-1}, z_l\) (thus also the edges \(e_1, e_2, \ldots, e_l\) are pairwise distinct.

**Definition 4.3.6** (Semi-Path). \(S\) is a semi-path \(P_s\) if and only if the vertices \(z_1, y_1, z_2, \ldots, y_{k}, y_{k+1}, \ldots, y_{l-1}, z_l\) (thus also the edges \(e_1, e_2, \ldots, e_l\) are pairwise distinct and \(y_k = y_{ij}\), and \(y_{k+1} = y_{ji}\) (\(y_k\) and \(y_{k+1}\) are considered as a same vertex).

**Definition 4.3.7** (Lasso). \(S\) is a lasso \(L\) if and only if the vertices \(z_1, y_1, \ldots, y_{l-1}, z_l, y_{l+1}\) and the edges \(e_1, e_2, \ldots, e_{l+1}\) are pairwise distinct and \(y_{l+1} = y_k\) for some \(k \in \{1, 2, \ldots, l\}\). A lasso can begin with a \(y\) node.

**Definition 4.3.8** (Semi-Lasso). \(S\) is a semi-lasso \(L_s\) if and only if the vertices \(z_1, y_2, \ldots, y_{l-1}, z_l, y_{l+1}\) and the edges \(e_1, e_2, \ldots, e_{l+1}\) are pairwise distinct and \(y_{l+1} = y_{ij}\) and there is a \(k \in \{1, 2, \ldots, l\}\) such that \(y_k = y_{ji}\). A semi-lasso can also begin with a \(y\) node.

**Definition 4.3.9** (A-alternating). A vertex-edge sequence (a walk) \(S = \{z_1, e_1, y_1, e_2, z_2, \ldots, y_{l-1}, e_l, z_l\}\) is A-alternating if and only if \(a(z_i) < 1 \ \forall i, i\) is odd, and \(a(z_i) > 0 \ \forall i, i\) is even, and \(z_i \in K\).

**Definition 4.3.10** (Modified A-alternating). A vertex-edge sequence (a walk) \(S = \{z_1, e_1, y_1, e_2, z_2, \ldots, y_{l-1}, e_l, z_l\}\) is modified A-alternating if and only if:

(i) \(a(z_i) < 1 \ \forall i, i\) is odd

(ii) \(a(z_i) > 0 \ \forall i, i\) is even, and \(z_i \in K\)

(iii) there is a pair \((y_i, z_{i+1})\) such that \(a(y_i) < 1 \ \text{if } a(z_i) < 1 \ \text{and } i\) is odd
Definition 4.3.11 (Augmenting Path). An A-alternating even length path
\( P = \{z_1, e_1, y_1, e_2, z_2, \ldots, y_{l-1}, e_l, z_l\} \) is augmenting if \( z_1 \) and \( z_l \) are A-unsaturated.

(a) Augmenting path starts and ends in a \( z \) node

(b) Augmenting path starts and ends in a \( y \) node

Figure 4.2: Example of augmenting paths

Definition 4.3.12 (Augmenting Semi-Path). An A-alternating even length semi-path \( P_s = \{z_1, e_1, y_1, e_2, z_2, \ldots, y_k, y_{k+1}, \ldots, y_{l-1}, e_l, z_l\} \) is augmenting if \( z_1 \) and \( z_l \) are A-unsaturated.

Figure 4.3: Example of semi-augmenting path

Definition 4.3.13 (Augmenting Lasso of Type I). An A-alternating lasso
\( L = \{z_1, e_1, y_1, \ldots, y_{l-2}, e_{l-1}, z_{l-1}, e_l, y_l\} \) is augmenting if

(i) \( z_1 \) and \( z_{l-1} \) are both A-unsaturated

(ii) \( l = 9 + 4k, \ \forall k = 0, 1, 2, 3, \ldots \)

(iii) \( y_l = y_k \) for some \( k \in \{1, 2, \ldots, l - 2\} \)

Figure 4.4: Example of augmenting lasso of Type I

Definition 4.3.14 (Augmenting Lasso of Type II). An A-alternating lasso
\( L = \{z_1, e_1, y_1, \ldots, y_{l-2}, e_{l-1}, z_{l-1}, e_l, y_l\} \) is augmenting if
(i) \(z_1\) is \(A\)-unsaturated

(ii) \(a(z_{l-1}) > 0\)

(iii) \(l = 7 + 4k, \ \forall k = 0, 1, 2, 3, \ldots\)

(iv) \(y_l = y_k\) for some \(k \in \{1, 2, \ldots, l - 2\}\)

**Figure 4.5: Example of augmenting lasso of Type II**

**Definition 4.3.15** (Augmenting Semi-Lasso of Type I). A (modified) \(A\)-alternating semi-lasso \(L_s = \{z_1, e_1, y_1, \ldots, y_{l-2}, e_{l-1}, z_{l-1}, e_l, y_l\}\) is augmenting if

(i) \(z_1, z_{l-1},\) and \(y_l\) are \(A\)-unsaturated

(ii) \(l = 9 + 4k, \ \forall k = 0, 1, 2, 3, \ldots\)

(iii) \(y_l = y_k\) for some \(k \in \{1, 2, \ldots, l - 2\}\)

**Figure 4.6: Example of augmenting semi-lasso of Type I**
Definition 4.3.16 (Augmenting Semi-Lasso of Type II). A (modified) $A$-alternating semi-lasso $L_s = \{z_1, e_1, y_1, \ldots, y_{l-2}, e_{l-1}, z_{l-1}, e_l, y_l\}$ is augmenting if

(i) $z_1$ is $A$-unsaturated

(ii) $a(z_{l-1}) > 0$, and $a(y_l) > 0$

(iii) $l = 7 + 4k$, $\forall k = 0, 1, 2, 3, \ldots$

(iv) $y_l = y_k$ for some $k \in \{1, 2, \ldots, l-2\}$

We define an augmenting structure in terms of primitive augmenting structures. Each of the augmenting structures we have defined so far is a primitive augmenting structure. In other words, an augmenting path, an augmenting semi-path, an augmenting lasso of Type I, an augmenting lasso of Type II, an augmenting semi-lasso of Type I, as well as an augmenting semi-lasso of Type II is a primitive augmenting structure. All these primitive augmenting structures can also begin or end (or both begin and end) with a $y$ node. For ease of presentation and readability, we did not include such structures that start and/or end with a $y$ node in the above definitions. However, the algorithm considers augmenting structures that begin and/or end with a $y$ node, and all the arguments for an augmenting structure that start and end with a $z$ node work for these cases as well.

Definition 4.3.17 (Augmenting Structure). An augmenting structure $S$ can be constructed recursively by appending an augmenting structure to a plus node of another augmenting structure. As the basis, any primitive augmenting structure is an augmenting structure. Formally:

(i) Any primitive augmenting structure is an augmenting structure.

(ii) A structure obtained by appending an augmenting structure to another augmenting structure at a plus node is an augmenting structure.
4.3.2 Combinatorial Algorithm for Linear Program of Linear Ordering Problem

We now informally outline the search process for an augmenting structure. We assume that there is a feasible solution $A(y, z)$ to the DRP. The search process relies on the optimal solution $A^*(y, z)$ to DRP, that satisfies the following property:

- Property I. $A^*$ minimizes $d(A, A^*) = \sum_y |a(y) - a^*(y)| + \sum_z |a(z) - a^*(z)|$

Algorithm $\text{FindAugmentation}(G)$ initiates the search and looks for any $A$-unsaturated leaf $z$ node, or any $A$-unsaturated node $y$. If it finds an $A$-unsaturated leaf node $z$, it calls $\text{AugmentStructure}(G, R, z, +)$; if it finds an $A$-unsaturated node $y$, it calls $\text{AugmentStructure}-(G, R, y, +)$. Algorithms $\text{AugmentStructure}(G, R, z, +)$ and $\text{AugmentStructure}-(G, R, y, +)$ first check if they can complete an augmenting structure. If they can, they stop the search process and return success (true). If they cannot complete an augmenting structure, they continue the search until either they can find an augmenting structure or report failure (return false).

The pseudocode for this search is given below in Algorithm $\text{FindAugmentation}(G)$. During the search, we refer to a node as either a plus node or a minus node. This is based on the value assigned to the node by the current solution in relation to the value assigned to the node by the optimal solution. However, we also refer to the sign of a node, which can either be $+$ or $-$.

Algorithm 6 Finding an augmenting structure

1: procedure $\text{FindAugmentation}(G)$
2: for any unsaturated $z$ which is also a leaf do
3: return $\text{AugmentStructure}(G, R, z, +)$
4: for any unsaturated $y$ do
5: return $\text{AugmentStructure}-(G, R, y, +)$

The Algorithm $\text{AugmentStructure}(G, z, s)$ starts at a $z$ node and takes different actions based on the sign of this $z$ node. If the sign is $+$, then it ensures that none of the constraints corresponding to the (unvisited) adjacent $y$ nodes get violated after the addition. To illustrate, suppose that we are in $z_1$ and the sign is $+$. The constraints corresponding to the adjacent $y$ nodes $y_1$ and $y_2$ are in danger of violation. The constraint for $y_1$ is of the form of $y_1 + z_1 + \sum_{z' \in N(y_1)/z_1} z' < 0$. After adding to $z_1$, the algorithm ensures that this constraint is not violated by subtracting from one of $z' \in N(y_1)/z_1$. To this end, it calls $\text{AugmentStructure}(G, z', -)$ for each node $z' \in N(y_1)/z_1$. The algorithm needs only one of these calls to return true. If one of these calls returns true, then we are sure that adding to $z_1$ does not violate any constraint. It can thus augment the current feasible solution. If the sign is $-$, then the algorithm verifies that $a(z) > 0$ (if this $z$ node is in set $K$). If however we subtract from the current $z$ node, none of the DRP constraints are in danger of violation. We only need to find a $y$ node such that we may continue the search process that
leads to an augmentation. During this search process, whenever the algorithm finds itself at a plus node, and the required sign for this node is also +, then adding to that node does not violate any constraint in DRP. Therefore, the algorithm stops and returns true.

We provide a more detailed description of the algorithm below. Algorithm \textsc{AugmentStructure}(G,z,s) first checks if the required sign is + or − in Line 2 and Line 29 (if condition). After confirming that s is +, in Line 3, the algorithm checks if the current node is a z leaf node; if it is, the algorithm stops and returns true. This indicates that the current branch under exploration ends at a plus node, and therefore it is augmenting. If the condition in Line 3 fails, the algorithm collects all the unvisited y nodes adjacent to the current z node under $T_z$ in Line 5. If the current z node is not a leaf and the number of y nodes adjacent to this node that are already visited is greater than 1, then the algorithm stops and returns true in Line 7. This happens only when there is an augmenting structure which is a lasso of Type I. If none of the conditions in Line 3 and Line 7 are true, then the algorithm continues by finding an adjacent y node such that it may continue the search process that leads to an augmentation (Lines 10-28). The for loop in Line 10 iterates on each $y \in T_z$ and collects the z nodes adjacent to y with $a^*(z) < a(z)$ (denoted $T_{y<}$) in Line 11 and Line 12. If $a^*(y) \geq a(y)$ fails on Line 14, then the algorithm tries to continue the search by finding a lasso or semi-lasso in Line 25, or by finding a semi-path or simple path in Line 27. In Lines 29-56, the algorithm executes steps similar to the above steps for s equals −.
Algorithm 7 Finding an augmenting structure

1. procedure AugmentStructure(G, z, s)
2.   if s = + then
3.     if z is a leaf node then
4.       return TRUE
5.     Tz = {y' ∈ N(z), y' is unvisited}
6.     if Tz > 1 then
7.       return TRUE
8.   f = TRUE
9.   fy = TRUE
10.  for every y ∈ Tz do
11.     Tz = {y′ ∈ N(y), y′ is unvisited}
12.     Tz = {z′ ∈ Tz ∧ a′(z) < a(z)}
13.     fy = TRUE
14.     if a′(y) ≥ a(y) then
15.       if Tz = φ then
16.         return TRUE
17.       f = FALSE
18.       for every z ∈ Tz do
19.         f = f ∨ AugmentStructure(G, z, −)
20.       fy = fy ∧ fy
21.     else
22.       f = FALSE
23.     for every z ∈ Tz do
24.       if z ∈ Tz then
25.         f = f ∨ AugmentStructure(G, z, −)
26.     else
27.       f = f ∨ AugmentStructure(G, z, −)
28.       fy = fy ∧ fy
29.   else
30.     if z is a leaf node then
31.       return FALSE
32.     Tz = {y′ ∈ N(z), y′ is unvisited}
33.     if Tz = φ then
34.       return TRUE
35.     f = TRUE
36.     fy = TRUE
37.     for every y ∈ Tz do
38.       Tz = {z′ ∈ N(y), z′ is unvisited}
39.       Tz = {z′ ∈ Tz ∧ a′(z) > a(z)}
40.       fy = TRUE
41.     if a′(y) ≤ a(y) then
42.       if Tz = φ then
43.         return FALSE
44.       f = FALSE
45.     for every z ∈ Tz do
46.       f = f ∨ AugmentStructure(G, z, +)
47.       fy = fy ∧ fy
48.   else
49.     f = FALSE
50.     for every z ∈ Tz do
51.       if z ∈ Tz then
52.         f = f ∨ AugmentStructure(G, z, +)
53.       else
54.         f = f ∨ AugmentStructure(G, z, +)
55.       fy = fy ∧ fy
56. return fy
The Algorithm AugmentStructure\((G,y,s)\) starts at a \(y\) node, and takes different actions based on the sign of this node. If the sign is +, then it will ensure that the constraint corresponding to the current \(y\) node does not get violated after the addition. For this purpose, it will call AugmentStructure\((G,z',-\)\) for each \(z' \in N(y)\), \(z'\) is unvisited. The algorithm needs only one of these calls to return true. If one of these calls returns true, then we are sure that adding to \(y\) does not violate its constraint and it also can augment the current feasible solution. If the sign is −, the algorithm either subtracts from one of the adjacent \(z' \in N(y)\) or subtracts from the current \(y\) node if \(y \in R\). Set \(R\) contains \(y\) nodes where both \(y_{ij}\) and \(y_{ji}\) are present in the DRP graph. Therefore, if \(y \in R\), the algorithm looks for a \(z \in N(\bar{y})\) and calls AugmentStructure\((G,z,+\)\) (note: if \(y = y_{ij}\), then \(\bar{y} = y_{ji}\)). If \(y \not\in R\), the algorithm looks for a \(z \in N(y)\) and calls AugmentStructure\((G,z,+\)\).

In Line 2, the algorithm checks the sign of \(s\). If the sign is +, the algorithm returns true in Line 4 if \(y_i \not\in R\). It indicates the current branch that ends in this \(y_i\) is augmenting. In Line 5, the algorithm looks for ending a semi-lasso by checking if \(y_i \in R\) and if \(\bar{y}_i\) is already visited. If \(y_i \in R\) and for all \(z \in N_{y_i}\), we have \(a^*(z) \leq a(z)\), then we can add to \(y_i\), but because \(y_i \in R\), we have to take care of the constraint corresponding to \(\bar{y}_i\). If this constraint get violated after the addition, then the algorithm calls AugmentStructure\((G,z,-\)\) on each \(z \in N(\bar{y})\) in Line 24, and then performs an ‘or’ of the results to make sure that the current solution stays feasible. If \(s = -\), \(y \not\in R\), and the algorithm is not able to subtract from any adjacent \(z \in T_{y_i}\), then it stops and returns false in Line 20 indicating the current branch under exploration is not augmenting. However, if it can subtract from any \(z \in T_{y_i}\), then it calls AugmentStructure\((G,z,-\)\) on each \(z \in N_{\bar{y}_i}\) and performs an ‘or’ of the results. If \(y_i \in R\) and \(\bar{y}_i\) are visited before, then the algorithm stops and returns true on Line 27, indicating it ends in a semi-lasso. If the condition for ending a semi-lasso fails on Line 26, then if \(y_i \in R\) and \(\bar{y}_i\) are not visited so far, the algorithm makes sure it continues the search process for an augmenting structure. The algorithm looks for each \(z' \in T_{\bar{y}_i} = \{z' | z' \in N(\bar{y}_i) \land a^*(z') < a(z'), z'\text{is unvisited}\}\) and call AugmentStructure\((G,z,+\)\) on each of them. This indicates that the algorithm subtractions from \(y_i\) (and also \(\bar{y}_i\)) and continues from the \(\bar{y}_i\)’s adjacent to \(z\) (semi-path). In Line 35, the results get ‘or’ed and stored in the \(f_{\bar{y}_i}\).
Algorithm 8 Finding an augmenting structure

1. procedure AugmentStructure(G,yi,s)
2. if s = + then
3. if yi ∉ R then
4. return TRUE
5. if yi ∈ R \ y̅i is visited before then
6. return TRUE
7. if yi ∈ R \ a*(zi) ≤ a(zi) \ ∀zi ∈ N(yi) then
8. T = {z|z ∈ N(yi), z is unvisited and has surplus}
9. if T = φ then
10. return TRUE
11. else
12. f = FALSE
13. for Every z′ in T do
14. f = f ∨ AugmentStructure(G,z′,−)
15. return f
16. else
17. if yi ∉ R then
18. Tyi = {z′|z′ ∈ N(yi) \ a*(z′) < a(z′), z′ is unvisited}
19. if Tyi = φ then
20. return FALSE
21. else
22. f̅y = FALSE
23. for every z′ ∈ Tyi do
24. f̅y = f̅y ∨ AugmentStructure(G,z′,−)
25. return f̅y
26. if yi ∈ R \ y̅i is visited before then
27. return TRUE
28. if yi ∈ R then
29. Tyi = {z′|z′ ∈ N(y̅i) \ a*(z′) ≤ a(z′), z′ is unvisited}
30. if Tyi = φ then
31. return FALSE
32. else
33. f̅y = FALSE
34. for every z′ ∈ Tyi do
35. f̅y = f̅y ∨ AugmentStructure(G,z′,+)
36. return f̅y

Algorithm AugmentStructure(G,yi,zi,s) is only called by Algorithm AugmentStructure-(G,zi,s) whenever there are adjacent y and z nodes that are either both plus nodes or minus nodes. If s = +, the algorithm initiates a search for a semi-lasso of Type I and continues the search in Line 12 by calling AugmentStructure(G,yi,zi,−). If s = −, the algorithm initiates a search for a semi-lasso of Type II and continues the search in Line 25 by calling AugmentStructure(G,yi,zi,+).
Algorithm 9 Finding an augmenting structure

1. procedure AugmentStructure($G,y_i,z_i,s$)
2. if $s = +$ then $\triangleright$ semi-lasso of type I
3. $f = TRUE$
4. $T_{z_i} = \{ y'|y' \in N(z_i), y' \text{ is unvisited} \}$
5. if $T = \emptyset$ then
6. return $TRUE$
7. else
8. for Every $y'$ in $T_{z_i}$ do
9. $f'_y = FALSE$
10. $T_{y'} = \{ z''|z'' \in N(y'), z'' \text{ is unvisited} \}$
11. for every $z''$ in $T_{y'}$ do
12. $f'_y = f'_y \lor$ AugmentStructure($G,z'',-)$
13. $f = f \land f'_y$
14. return $f$
15. else $\triangleright$ semi-lasso of type II
16. $f = FALSE$
17. $T_{z_i} = \{ y'|y' \in N(z_i), y' \text{ is unvisited} \}$
18. if $T = \emptyset$ then
19. return $TRUE$
20. else
21. for Every $y'$ in $T_{z_i}$ do
22. $f'_y = FALSE$
23. $T_{y'} = \{ z''|z'' \in N(y'), z'' \text{ is unvisited} \}$
24. for every $z''$ in $T_{y'}$ do
25. $f'_y = f'_y \lor$ AugmentStructure($G,z'',+)$
26. $f = f \lor f'_y$
27. return $f$

Lemma 8. Algorithm FindAugmentation($G$) will terminate.

Proof. Algorithm FindAugmentation($G$) will either call AugmentStructure($G,R,z,+)$ or AugmentStructure($G,R,y,+)$). Both these algorithms stop and return true or false whenever they revisit a node. In the worst case, Algorithm FindAugmentation($G$) visits all the nodes of the DRP graph.

Lemma 9. The feasible solution $A(y,z)$ remains feasible during the execution of Algorithm FindAugmentation($G$).

Proof. Algorithm FindAugmentation($G$) starts with the feasible solution $A(y,z)$. We show that this solution remains feasible during the execution of the algorithm. FindAugmentation($G$) will either call AugmentStructure($G,R,z,+)$ or AugmentStructure($G,R,y,+)$). In either case, when $z$ is a plus node ($a^+(z) > a(z)$), we add to the current $a(z)$ to augment the current solution. Since node $z$ can participate in at most three different constraints in the DRP, we are in danger of violating the feasibility of our solution after such addition. Let $y_1, y_2$, and $y_3$ be the $y$ nodes
adjacent to the current $z$ node. The corresponding constraints for these $y$ nodes are:

\[
y_1 + z + \sum_{z' \in N(y_1)/z} z' < 0
\]
\[
y_2 + z + \sum_{z' \in N(y_2)/z} z' < 0
\]
\[
y_3 + z + \sum_{z' \in N(y_3)/z} z' < 0
\]

Figure 4.8: Subgraph corresponds to constraints for $y_1, y_2,$ and $y_3$

Suppose that we have already visited $y_1$ and we reach the current $z$ node from this node. In this case, only constraints corresponding to $y_2$ and $y_3$ are in danger of violation. Consider the constraint $y_2 + z + \sum_{z' \in N(y_2)/z} z' < 0$. To stay feasible, we have to subtract either from $y_2$ or subtract from one of the $z'$ nodes adjacent to $y_2$. The same needs to be done for $y_3$. This corresponds to the ‘and’ in the algorithm. During the execution of the algorithm, the optimal solution guides us in augmenting our non-optimal solution. If $z$ is a minus node ($a^*(z) < a(z)$), we subtract from the current $z$. In this case, we only need to add to either $y_2$ or $y_3$ (or to one of the $z'$ nodes adjacent to each of these $y$ nodes). This corresponds to the ‘or’ in the algorithm. When $y$ is a plus node ($a^*(y) > a(y)$), we can add to the current $a(y)$ in order to augment the current solution. In this case, the constraint $y + \sum_{z' \in N(y)} z' < 0$ is in danger of violation after such an addition. To remain feasible, we only need to subtract from one of the $z'$ nodes adjacent to this $y$ node to remain feasible. This corresponds to the ‘or’ in the algorithm.

When $y$ is a minus node ($a^*(y) < a(y)$), we can either subtract from the current $y = y_{ij}$ and $\bar{y} = y_{ji}$, and add to one of the nodes $z \in N(\bar{y})$ (augmenting semi-path) or subtract from $y$ and one of the adjacent $z$ nodes (augmenting semi-lasso of Type II or augmenting lasso of Type II).

Lemma 10. If Algorithm FindAugmentation($G$) returns false, then $A(y, z)$ is optimal.

Proof. We prove the contrapositive statement: if $A(y, z)$ is not optimal, then FindAugmentation($G$) returns true. Assume $A(y, z)$ is not optimal and Algorithm FindAugmentation($G$) returns false. We will arrive at a contradiction.
Since we assume that algorithm \textsc{FindAugmentation}(G) returns false, there is at least one path in S starting from v (a plus node) that ends in a minus node. We call this path P. We now show that by subtracting a positive quantity \( g \) from the plus nodes in path P, and adding the same quantity to the minus nodes, we can obtain from \( x^*(y,z) \) another solution \( \tilde{x}^*(y,z) \) with the same value but which is closer to our feasible solution \( x(y,z) \). However, this is a contradiction, since we assume that \( x^*(y,z) \) satisfies Property I. We call this procedure of subtracting and adding (the same positive amount \( g \)) to plus and minus nodes ‘flipping’.

During the execution of algorithm \textsc{FindAugmentation}(G), whenever we have a minus node (be it a \( z \) node or a \( y \) node), we have to find only one outgoing path from this node that ends up in augmenting the current solution. This corresponds to the ‘or’ operation in the pseudocode. To show that by flipping we can obtain an optimal solution closer to our feasible solution, we have to worry about the \( z \) minus nodes. After adding the quantity \( g \) to the optimal solution \( x^*(z) \), we are in danger of losing the feasibility of the optimal solution (violating one of the constraints this \( z \) node participates in). If the degree of the minus \( z \) node (in the original graph) that we add to is less than or equal to 2, we are safe (Structure S is alternating, and we also subtract from the adjacent plus node(s)).

However if the degree of the \( z \) node is 3 (in the original graph), then in order for \( x^*(y,z) \) to stay feasible, we have to explore both outgoing edges from this \( z \) node and ensure that \( \tilde{x}^*(y,z) \) stays feasible after flipping.

We now show that if \textsc{FindAugmentation}(G) returns false, we can modify the optimal solution \( x^*(y,z) \), and obtain another optimal solution \( \tilde{x}^*(y,z) \) which is closer to our feasible solution \( x(y,z) \).

**Case I.** All the \( z \) minus nodes in structure S have degree less than or equal to 2 (in the original graph). In this case, we can easily subtract quantity \( g \) from the plus nodes and add the same amount to the minus nodes without violating the feasibility of \( \tilde{x}^*(y,z) \) according to Lemma 9.

**Case II.** There are some \( z \) minus nodes in the structure S return by \textsc{FindAugmentation}(G) which have degree 3 (in the original graph).

Suppose that \( Z_3 = \{z_{i1}, z_{i2}, \ldots, z_{ir} \} \) are the minus \( z \) nodes that have degree 3 in the original graph. Again we are going to do the flipping. However for every minus \( z \in Z_3 \), after adding quantity \( g \), we have to explore the other outgoing structure which is not a part of S. We show that after exploring this added structure, the objective function value of \( v(x^*) \) cannot change. Let us assume that we add a quantity \( g \) to node \( z_i \in Z_3 \) and we explore the other path which is not in the structure S. Let us call the structure explored after this operation \( S' \) (including \( z_i \)). Let \( N_{S'} \) denote the total net augmentation of \( S' \) after the flipping. There are three cases:

**Case II.a.** \( N_{S'} > 0 \). We can modify the optimal solution \( x^*(y,z) \) by adding a quantity \( g \) to the minus \( z \) nodes and subtracting the same amount from the plus \( z \) nodes in structure S. We can continue to do this along structure \( S' \), and obtain another solution \( \tilde{x}^*(y,z) \) where \( v(\tilde{x}^*(y,z)) > v(x^*(y,z)) \). This contradicts the optimality of \( x^*(y,z) \).
Case II.b. $N_{S'} = 0$. We can add a quantity $g$ to the minus $z$ nodes and subtract the same amount from plus $z$ nodes in the optimal solution $A^*(y, z)$. We can continue to do this along structure $S'$, and obtain another solution, $\bar{A}^*(y, z)$ with the same objective function value. However, $\bar{A}^*(y, z)$ has less distance to our solution feasible solution $A(y, z)$, which contradicts Property I.

Case II.c. $N_{S'} < 0$. Instead of exploring structure $S$ at node $z_i$ (which did not result in an augmenting structure), we could have explored structure $S'$ and augmented the feasible solution $A(y, z)$. However, the following two cases could have prevented us from exploring $S'$:

i There is a $z$ node in the structure $S'$ such that $a(z) = 1$. This $z$ node was a plus node before the flipping procedure (adding a quantity $g$ to $z_i$). Since $z$ is at its maximum allowed capacity, it is not possible to add anymore to $a(z)$. Since the path that leads to $z$ is alternating, this path should be of the form \( \ldots, z', y', z, \) where $z'$ is a minus node, and $z$ is a plus node. We know that because of the capacity constraint we could not add to $z$. However, we could have added to $y'$ immediately after subtracting from $z'$. The only reason that the search could not add to $y'$ was because $y'$ was also at its maximum capacity ($a(y') = 1$). To summarize, we have $a^*(z') < a(z')$, and $a(z) = a(y') = 1$. Let us consider the constraint that corresponds to $y', a(y') + a(z') + a(z) \leq 0$, and substitute the values for $z, y'$. We get $a(z') \leq -2$. Since $z'$ is a minus node, $a^*(z') < a(z')$. Thus it follows that $a^*(z') < -2$. This contradicts the optimality of $A^*(y, z)$ since we could have added to $z'$ without violating any constraints.

ii There is a $z$ node in structure $S'$ such that $a(z) = 0$ and $z \in K$. This $z$ node was a minus node ($a(z) > a^*(z)$) before the flipping procedure, which adds a quantity $g$ to $z_i$. Since $a(z) = 0$, then $a(z) > a^*(z)$ cannot be true unless $A^*(y, z)$ was infeasible. We conclude that it is not possible to have a minus $z$ node with $a(z) = 0$ in an alternating path.

We conclude that neither of the above cases i and ii could have prevented the search procedure from exploring $S'$. Thus the search procedure would have also explored $S'$, which leads to an augmenting path. This is true for every path $S$ that ends in a minus node, contradicting the optimality of $A^*(y, z)$.

Lemma 11. If $\text{FindAugmentation}(G)$ returns true, there might be an $A$-augmenting structure.

Proof. Algorithm $\text{FindAugmentation}(G)$ starts with a plus node, say $s$ (s may either be an $A$-unsaturated leaf node $z$ or an $A$-unsaturated $y$ node) and returns a structure $S$. The algorithm adds a quantity $e$ to the plus nodes and subtracts the same quantity from the minus nodes. For a plus $z$ node, the algorithm ensures that none of the constraints that this $z$ node participates in gets violated. This corresponds to the `and` operation in the pseudocode. Basically at each plus $z$ node the algorithm waits for both (at most two) outgoing branches to return true. When a branch returns true, it means two things. First, none of the constraints that correspond to the nodes on
that branch get violated after the augmentation. Second, the branch ends in a plus node (potential augmentation). If one of the branches returns a false, either at least one constraint corresponding to a \( y \) node in that branch is infeasible, or the branch ends in a minus node. Therefore the structure \( S \) is not augmenting. In each minus node (be it a \( z \) node or a \( y \) node), the algorithm waits for at least one of the outgoing branches to return true. This corresponds to the ‘or’ operation in the pseudocode. If none of the branches return true, then the structure \( S \) is not augmenting.

When algorithm \( \text{FindAugmentation}(G) \) returns true, it indicates two things. First, adding a quantity \( \epsilon \) to the plus nodes and subtracting the same quantity from the minus nodes does not violate any constraint in the DRP. Second, every branch starting from the starting node \( t \) returns true. At this point, we can call Algorithm \( \text{ComputeFlow}(S,v,count\_array,p) \). If the value returned by Algorithm \( \text{ComputeFlow}(S,v,count\_array,p) \) is positive, it means the net augmentation is positive and we can augment structure \( S \). On the other hand, if the net augmentation is 0, then structure \( S \) is not augmenting.

During the execution of Algorithm \( \text{AugmentStructure}(G) \), we keep information indicating whether the node is a plus node or a minus node, as well as the type of augmenting structure obtained. We use this information in Algorithm \( \text{ComputeFlow}(S,v,count\_array,p) \) to verify if a structure \( S \) returned by Algorithm \( \text{AugmentStructure}(G) \) is augmenting or not. The variable \( v \) denotes the starting vertex, and \( count\_array \) is an array of large size with all entries equal to 0 at the beginning of Algorithm \( \text{ComputeFlow}(S,v,count\_array,p) \). The initial value of \( p \) is also set to \( 2^0 \). This refers to the fact that we start by adding or subtracting \( \frac{\epsilon}{2} \) to the nodes in the augmenting path. Algorithm \( \text{AugmentStructure}(G) \) starts with a flow of \( \epsilon \) and adds this amount to the plus nodes and subtracts it from the minus nodes. Whenever we visit a node from which the cycle of a lasso or semi-lasso (of either type) starts, the amount that we add to any plus node (subtract from any minus node) from this point onward is divided by 2 (i.e., the amount is \( \epsilon/2 \)). The variable \( p \) keeps track of the fraction of \( \epsilon \) that we add to (or subtract from) nodes at any time during the traversal of structure \( S \). During the traversal of structure \( S \), when we visit a plus node, we add 1 to the current entry of \( count\_array[p] \), and when we visit a minus node, we subtract 1 from the current entry of \( count\_array[p] \). At the end of the traversal, if the sum of entries of \( count\_array \) is strictly positive, it indicates that structure \( S \) is augmenting, otherwise it is not.
Algorithm 10 Compute the flow

1: procedure ComputeFlow(S,v,count_array,p)
2:     if v is y node then
3:         if v.sign = + then \(\triangleright\) plus y node
4:             count_array[p] = count_array[p] + 1
5:         if v.status = semi_lasoo then
6:             p = p/2
7:             count_array[p] = count_array[p] - 1
8:     else if v.sign = − then \(\triangleright\) neutral y node
9:         count_array[p] = count_array[p] + 1
10:        if v.reverse = True then
11:            count_array[p] = count_array[p] + 1
12:       else
14:          if v.reverse = True then
15:             count_array[p] = count_array[p] + 1
16:           else
17:               if v.status = lasso then
18:                   p = p/2
19:               N_v = \{v′ ∈ N(v), v′ is unvisited\}
20:           else
21:               return
22:         for every t ∈ N_v do
23:             ComputeFlow(S,t,count_array,p)
24:       else \(\triangleright\) minus z node
26:        if N_v = φ then
27:            return
28:       for every t ∈ N_v do
29:           ComputeFlow(S,t,count_array,p)
30:     else
31:         count_array[p] = count_array[p] + 1
32:        if N_v = φ then
33:           return
34:       for every t ∈ N_v do
35:         ComputeFlow(S,t,count_array,p)
36:     else \(\triangleright\) plas z node
38:        if N_v = φ then
39:           return
40:       for every t ∈ N_v do
41:         ComputeFlow(S,t,count_array,p)
42:     return
43: end procedure

Theorem 12. Let \(A(y,z)\) be a solution to the DRP that is not maximum. Then there is a \(A\)-augmenting structure.

Proof. Follows from Lemma 8 to 11.  \(\square\)
Chapter 5

Cyclical Shift Scheduling

5.1 Introduction

The cyclical scheduling problem is used to schedule shifts for workers in a factory. Given a set of \( n \) work periods, each worker is assigned a shift where he works for \( n - 2 \) consecutive periods, and takes off the remaining 2 periods. Thus, for \( n = 7 \), a typical shift may be to work from Monday to Friday and take off Saturday and Sunday. Another shift may be to work from Friday to Tuesday and take off Wednesday and Thursday (there are 7 such shifts in a week). Each shift may also have a cost associated with it. In addition, the factory requires that a given number of workers be available each period (this requirement may vary from period to period). The objective is to assign a shift to each worker such that the daily requirement is fulfilled and the total cost of the shifts is minimized. We solve the cyclical scheduling problem by solving a series of \( b \)-matching problems.

Tibrewala et. al [85] provide an integer programming formulation for the problem when all shift costs are equal. They also provide a simple algorithm to solve the problem optimally. Bartholdi and Ratliff [13] solve the cyclical scheduling problem by considering the ‘complementary problem’ where given the number of workers \( w \) in the factory, they maximize the number of workers who are off for two consecutive periods. There is an upper bound on the number of workers who are off during each period. The number of workers they start out with, \( w \), is adequate if the objective function value is at least as much as \( w \). A binary search procedure is used to find the minimum value for \( w \) such that \( w \) is adequate.

Bartholdi et. al [12] extend this approach to the general \((k,n)\) cyclic scheduling problem, where there is a total of \( n \) periods, and each worker is assigned a shift of \( k \) consecutive periods to work (and is off for the remaining \((n-k)\) periods). There is a cost associated with each shift, and the objective is to minimize the total cost of shifts assigned to workers. They provide a parametric solution to this general problem by solving a series of network flow problems, one for each guess for the workforce size. In addition, they also provide a solution using linear programming and roundoff. Using the above notation, we address the \((n-2,n)\) cyclic scheduling problem in this research. Alfares [3]
provides computational results using linear programming for the (5, 7) cyclic scheduling problem. Alfares [4] proposes an integer programming formulation for the cyclic scheduling problem where each worker gets 3 days off each week, with additional constraints, such as having at least 2 of the 3 off days consecutive. In a more recent paper, Hochbaum and Levin [50] provide a quadratic algorithm when all shift costs are equal. Hochbaum and Levin [49] generalize the cyclical scheduling problem where each worker is assigned multiple shifts, where each shift is a sequence of consecutive work periods followed by a sequence of off periods. They show that this problem is NP-hard when there are two or more such shifts, and propose approximation algorithms for the problem.

The very first polynomial time algorithm for finding the maximum size \( b \)-matching was proposed by Johnson [56]. It was an extension of Edmond’s algorithm [40]. Gabow [44] also provided an \( O(nm \log n) \) time algorithm for finding the maximum size \( b \)-matching (\( n \) is the number of vertices and \( m \) is the number of edges). The \( b \)-matching polytope (convex hull) of a graph was characterized by Edmonds and Pulleyblank [77]. Pulleyblank [76] has also provided a pseudo polynomial algorithm to find a maximum weighted \( b \)-matching. The first polynomial time algorithm to solve optimal maximum weighted \( b \)-matching was given by Cunningham and Marsh [32]. Marsh [66] also extended the Edmond’s weighted matching algorithm for finding the maximum weighted \( b \)-matching. Anstee [8] and Gerard [46] found strongly polynomial algorithms for this problem.

5.2 The \( b \)-Matching Formulation

We provide the formulation for the cyclical scheduling problem with \( n \) periods, with each shift comprising \( n - 2 \) consecutive work periods and 2 off periods. Shift \( s_i \) assigns off periods \( i \) and \( (i + 1) \mod n \) and the remaining as working periods. \( x_i \) denotes the number of workers assigned to shift \( s_i \) and \( c_i \) denotes its cost. \( b_i \) denotes the number of workers required to work during period \( i \). The total cost of the assignment is given by \( \sum_{i=0}^{n-1} c_i x_i \). The left hand side of each constraint is the number of workers working on shift \( s_i \), and is given by \( x_{(i+2) \mod n} + x_{(i+3) \mod n} + \cdots + x_{(i-1) \mod n} \). This should be no less than the requirement \( b_i \). Without loss of generality, we assume all index arithmetic is done \( \mod n \) in the formulations for Problem I and Problem II that follow:

\[
\begin{align*}
\text{Problem I} & \quad \text{Minimize} \quad S = \sum_{i=0}^{n-1} c_i x_i \\
\text{subject to} \quad & \quad x_{i+1} + x_{i+2} + \cdots + x_{i-2} \geq b_i \quad \forall i, 0 \leq i \leq n - 1 \\
& \quad x_i \geq 0 \text{ and integer} \quad \forall i, 0 \leq i \leq n - 1
\end{align*}
\]

The above can be reformulated as a \( b \)-matching problem. If \( w = \sum_{i=0}^{n-1} x_i \) denotes the total number of workers, the number of workers working on shift \( s_i \) during period \( i \) is given by \( w - x_{i-1} - x_i \).
Letting \( d_i = w - b_i \), each of the above inequalities can be rewritten as \( x_{i-1} + x_i \leq d_i \), where \( d_i \) denotes the number of workers that are off during period \( i \). We thus obtain the formulation for Problem II below, given by Bartholdi and Ratliff [13]:

\[
\text{Problem II} \\
\begin{align*}
\text{Maximize} & \quad S = \sum_{i=0}^{n-1} c_i x_i \\
\text{s.t.} & \quad x_{i-1} + x_i \leq d_i \quad \forall i, 0 \leq i \leq n - 1 \\
& \quad x_i \geq 0 \text{ and integer} \quad \forall i, 0 \leq i \leq n - 1
\end{align*}
\]

The term \( c_i x_i \) in the objective function corresponds to the cost that is saved in period \( i \) because \( x_i \) workers in shift \( s_i \) are off during this period. The objective function thus maximizes the total cost saved over the \( n \) periods. The above formulation for Problem II can be visualised as a graph \( G = (V,E) \) with vertex set \( V = \{v_0, v_1, \ldots, v_{n-1}\} \) and edge set \( E = \{e_0, e_1, \ldots, e_{n-1}\} \), where \( e_0 = (v_0, v_1) \), \( e_1 = (v_1, v_2) \), \ldots, \( e_{n-1} = (v_{n-1}, v_0) \). We associate capacities \( d_0, d_1, \ldots, d_{n-1} \) with the vertices \( v_0, v_1, \ldots, v_{n-1} \) respectively.

We describe in this chapter an algorithm based on augmenting paths to solve the above problem. We start with an estimate for \( w \), the total number of workers. If \( \bar{w} = \sum_{i=0}^{n-1} \bar{x}_i < w \), where \( \bar{x}_i \) is the optimal solution to Problem II, and \( w \) is our estimate of the number of workers, then we revise this estimate upward. As observed by Bartholdi and Ratliff [13], a binary search may be performed to find the smallest value of \( w \) for which \( \bar{w} \geq w \). By solving Problem II for this value of \( w \), we obtain the optimal assignment of shifts to workers.

If \( w^* \) is the optimal value of \( w \) returned by the binary search procedure, then \( b_{\text{max}} \leq w^* \leq \sum_{i=0}^{n-1} b_i \), where \( b_{\text{max}} = \max_i \{b_i\} \). Bartholdi and Ratliff [13] also prove the following inequalities for a particular value \( w = w' \) and its corresponding solution \( \{x_i\} \):

\[
\sum_{i=0}^{n-1} x_i < w' \iff w' < w^* \quad (5.1)
\]
\[
\sum_{i=0}^{n-1} x_i \geq w' \iff w^* \leq w' \quad (5.2)
\]

They then provide the following simple algorithm to find \( w^* \).

Step 1  Restrict \( w^* \) within the interval \( b_{\text{max}} \leq w^* \leq \sum_{i=0}^{n-1} b_i \) where \( w^* \) is an integer.

Step 2  Perform binary search in this interval to locate \( w^* \). At each iteration \( w \) is fixed at a value \( w' \), and the corresponding version of Problem II is solved; the assignment \( x_i \) for each \( i \) is obtained from the optimal solution to Problem II, and \( \sum_{i=0}^{n-1} x_i \) is compared to \( w' \) and checked using inequalities (5.1) and (5.2) to further restrict the location of \( w^* \) to \( w' < w^* \) or \( w^* \leq w' \).
For the rest of this chapter, we focus on an efficient solution to Problem II, the $b$-matching problem. We provide an $O(n \log n)$ algorithm to solve the $b$-matching problem. The solution to the $b$-matching problem may be used in conjunction with the binary search procedure above to obtain a solution to the cyclical scheduling problem (Problem I) in $O(n \log n (\log nb_{\text{max}}))$ time.

To describe the algorithm for the $b$-matching problem, we first define saturated vertices, unsaturated vertices, alternating paths, and augmenting paths. All the above definitions are with respect to a feasible solution $x = \{x_0, x_1, \ldots, x_{n-1}\}$.

### 5.3 Definitions

Without loss of generality, all index arithmetic in the definitions below are done mod $n$.

**Definition 5.3.1.** A vertex $v_i$ is **saturated** if $x_{i-1} + x_i = d_i$. It is **unsaturated** otherwise.

**Definition 5.3.2.** A path is a sequence of vertices and edges $v_i, e_i = (v_i, v_{i+1}), v_{i+1}, \ldots, e_{j-1} = (v_{j-1}, v_j), v_j$, where no vertex or edge is repeated. The length of a path is the number of edges in it. Alternatively, the above path can also be described as the reversed sequence of vertices and edges $v_j, e_{j-1} = (v_j, v_{j-1}), \ldots, v_{i+1}, e_i = (v_{i+1}, v_i), v_i$.

**Definition 5.3.3.** Alternating Path of Type I: The path from $v_i$ to $v_j$ (from $v_j$ to $v_i$) is an alternating path of Type I if the feasible solution $x$ assigns a strictly positive value to edges that occur at even positions, namely edges $e_{i+1}, e_{i+3}, \ldots$ ($e_{j-2}, e_{j-4}, \ldots$).

**Definition 5.3.4.** Alternating Path of Type II: The path from $v_i$ to $v_j$ (from $v_j$ to $v_i$) is an alternating path of Type II if the feasible solution $x$ assigns a strictly positive value to edges that occur at odd positions, namely edges $e_i, e_{i+2}, \ldots$ ($e_{j-1}, e_{j-3}, \ldots$).

**Definition 5.3.5.** Odd Length Augmenting Path of Type I: An odd length alternating path of Type I from $v_i$ to $v_j$ (from $v_j$ to $v_i$) is augmenting if it is of odd length, vertices $v_i$ and $v_j$ are unsaturated, intermediate vertices are saturated, and $\sum_{e \text{ at odd position}} c_e > \sum_{e \text{ at even position}} c_e$.

![Figure 5.1: Example of odd length augmenting path of Type I](image)

**Definition 5.3.6.** Even Length Augmenting Path of Type I: An even length alternating path of Type I is augmenting if it is of even length, vertex $v_i$ is unsaturated, vertices $v_{i+1}, v_{i+2}, \ldots, v_{j-1}$ are saturated, and $\sum_{e \text{ at odd position}} c_e > \sum_{e \text{ at even position}} c_e$. 

57
Definition 5.3.7. Odd Length Augmenting Path of Type II: An odd length alternating path of Type II from \( v_i \) to \( v_j \) (from \( v_j \) to \( v_i \)) is augmenting if it is of odd length, all intermediate vertices \( v_{i+1}, v_{i+2}, \ldots, v_{j-1} \) are saturated, and \( \sum_{e \text{ at even position}} c_e > \sum_{e \text{ at odd position}} c_e \).

Definition 5.3.8. Even Length Augmenting Path of Type II: An even length alternating path of Type II from \( v_i \) to \( v_j \) is augmenting if it is of even length, all intermediate vertices \( v_{i+1}, v_{i+2}, \ldots, v_{j-1} \) are saturated, vertex \( v_j \) is unsaturated, and \( \sum_{e \text{ at even position}} c_e > \sum_{e \text{ at odd position}} c_e \).

We note that an even length augmenting path of Type I from vertex \( v_i \) to \( v_j \) is also an even length augmenting path of Type II from vertex \( v_j \) to vertex \( v_i \). Also, an even length alternating path of Type I (Type II) may be augmenting if it is considered from \( v_i \) to \( v_j \), but may not be augmenting if it is considered from \( v_j \) to \( v_i \). Finally, an augmenting path of Type I (Type II) may either be an odd length augmenting path of Type I (Type II), or an even length augmenting path of Type I (Type II).

5.4 A Simple Augmentation Algorithm

In this section, we outline a simple augmentation algorithm to find the optimal solution to the \( b \)-matching problem. The algorithm continues to augment the solution as long as an augmenting path of either type exists with respect to the solution. For an augmenting path of Type I, the algorithm augments the solution by adding to (subtracting from) each odd (even) edge. For a Type II augmenting path, it augments the solution by adding to (subtracting from) each even (odd) edge.
The algorithm terminates with the optimal solution when there is no augmenting path of either type with respect to a solution. The algorithm below does not specify how an augmenting path, or what type of augmenting path, is obtained.

**Algorithm 11 Algorithm Simple Augmentation**

1. Initialize $x_i = 0$ for $i = 0, 1, \ldots, n - 1$.
2. while there is an augmenting path do
   3. Obtain an augmenting path $p$ from $v_i$ to $v_j$. For odd (even) length augmenting path of Type I, let $\delta = \min\{x_e | e \text{ is even edge}, d_i - x_i - x_{i-1}, d_j - x_j - x_{j-1}\}$ ($\delta = \min\{x_e | e \text{ is even edge}, d_i - x_i - x_{i-1}\}$). Augment solution by adding $\delta$ to each odd edge, and subtracting $\delta$ from each even edge. For odd (even) length augmenting path of Type II, let $\delta = \min\{x_e | e \text{ is odd edge}\}$ ($\delta = \min\{x_e | e \text{ is odd edge}, d_j - x_j - x_{j-1}\}$). Augment solution by subtracting $\delta$ from each odd edge, and adding $\delta$ to each even edge.

We now prove the correctness of the algorithm by showing that if there are no augmenting paths with respect to a feasible solution $x = <x_0, x_1, \ldots, x_{n-1}>$, then $cx$ is the optimal solution value.

**Theorem 13.** If there are no augmenting paths with respect to a feasible solution $x$, then $cx$ is maximum.

*Proof.* We prove by contradiction. Assume that $cx$ is not optimal. Let $x^*$ be the optimal solution. Let $e_i$ be an edge such that $x_i < x^*_i$ (such an edge must exist). We consider the following three cases:

**Case I:** Vertices $v_i$ and $v_{i+1}$ are not saturated with respect to $x$. This implies the existence of an augmenting path (of Type I), which is a contradiction.

**Case II:** Either vertex $v_i$ or vertex $v_{i+1}$ is saturated (the other is unsaturated). Without loss of generality, let $v_{i+1}$ be saturated (and $v_i$ unsaturated). $x_i < x^*_i$ implies that $x_{i+1} > x^*_{i+1}$. We now construct an even length path $p$ of Type I which starts at unsaturated vertex $v_i$ goes through vertex $v_{i+1}$, and ends at vertex $v_{i+2}$. We can add $\delta > 0$ to edge $x_i$ and subtract $\delta$ from edge $x_{i+1}$ to construct a solution vector which is closer to the optimal solution vector. Path $p$ is thus an even length augmenting path of Type I, leading to a contradiction.

**Case III:** Both vertex $v_i$ and vertex $v_{i+1}$ are saturated. $x_i < x^*_i$ implies that $x_{i+1} > x^*_i$ and $x_{i-1} > x^*_{i-1}$. In this case, we construct an odd length path $p$ of Type II which starts at vertex $v_{i-1}$, goes through vertices $v_i$ and $v_{i+1}$, and ends at vertex $v_{i+2}$. We can subtract $\delta > 0$ from edge $x_{i-1}$, add $\delta$ to edge $x_i$, and subtract $\delta$ from edge $x_{i+1}$ to construct a solution vector which is closer to the optimal solution vector. Path $p$ is thus an odd length augmenting path of Type II, leading to a contradiction.

To derive an efficient algorithm for the $b$-matching problem, we next derive some properties of augmenting paths. In the next section, we present a modified algorithm for the $b$-matching problem.
This algorithm chooses at each stage an augmenting path of largest cost. We show that such an augmenting path is an odd length augmenting path of Type I.

5.5 Properties of Augmenting Paths

For any augmenting path of Type I (Type II), an edge at an odd (even) position is called a plus edge, and an edge at an even (odd) position is called a minus edge. The cost of an augmenting path $p$, denoted $c(p)$ is given by $\sum_{e \text{ is a plus edge}} c_e - \sum_{e \text{ is a minus edge}} c_e$. Note that our objective is to maximize the total cost of the matching, and therefore using an augmenting path $p$ with cost $c(p)$ to modify the matching increases the objective function value by $c(p)$. We now describe a modified algorithm which at each stage chooses only an augmenting path which has the largest cost.

<table>
<thead>
<tr>
<th>Algorithm 12 Algorithm Large-cost Augmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: Initialize $x_i = 0$ for $i = 0, 1, \ldots, n - 1$</td>
</tr>
<tr>
<td>2: while there is an augmenting path do</td>
</tr>
<tr>
<td>3: Search for an augmenting path $p$ with the largest cost. Let path $p$ start at vertex $v_i$ and end at vertex $v_j$. $p$ is an odd length augmenting path of Type I. Let $\delta = \min{x_e</td>
</tr>
</tbody>
</table>

We will now show that an augmenting path of largest cost is always an odd length augmenting path of Type I. We first show in Lemma 14 below that at any stage of Algorithm Large-cost Augmentation, an augmenting path of largest cost cannot be an odd length augmenting path of Type II.

**Lemma 14.** At any stage of Algorithm Large-cost Augmentation, an odd length augmenting path of Type II cannot have the largest cost.

**Proof.** We provide a proof by induction.

**Base Case:** There are no length 3 augmenting paths of Type II that have the largest cost.

We show by contradiction that an odd length augmenting path of Type II with length 3 cannot have the largest cost. Without loss of generality, let such a path $p$ start at vertex $v_i$ and end at vertex $v_{i+3}$, and go through vertices $v_{i+1}$ and $v_{i+2}$. Also, let $c_{i+1} > c_i + c_{i+2}$. Its existence implies that $x_i > 0$ and $x_{i+2} > 0$. Since vertices $v_{i+1}$ and $v_{i+2}$ are saturated, the edges $e_i$ and $e_{i+2}$ must have been augmented in earlier iterations, which is a contradiction given that edge $e_{i+1}$ has higher cost and would have been augmented before edges $e_i$ and $e_{i+2}$ get augmented. Furthermore, edge $e_{i+1}$ would get augmented until either one or both of vertices $v_{i+1}$ and $v_{i+2}$ get saturated. If $v_{i+1}$ gets saturated, then $x_i = 0$ and if $v_{i+3}$ gets saturated, then $x_{i+2} = 0$. In either case, such a path $p$ would not exist.

**Induction Hypothesis:** There are no odd length augmenting paths of Type II of length $k$ or less.
**Induction Step:** Consider an odd length augmenting path $p$ of Type II with length $k + 1$. Without loss of generality, let the path start at vertex $v_i$ and end at vertex $v_l$. The induction hypothesis implies that odd length subpaths $p_1$ of Type I from $v_i$ to an intermediate vertex $v_j$ and $p_3$ from vertex $v_l$ to an intermediate vertex $v_k$ were augmented in earlier iterations. In addition, the cost of subpath $p_2$ from intermediate vertex $v_j$ to intermediate vertex $v_k$ exceeds the costs of subpaths $p_1$ and $p_3$. Thus, subpath $p_2$ would get augmented before subpaths $p_1$ and $p_3$. Also, subpath $p_2$ would get augmented until either one or both of vertices $v_j$ and $v_k$ get saturated, or a minus edge gets empty. If vertex $v_j$ gets saturated, then subpath $p_1$ would not get augmented and if vertex $v_k$ gets saturated, then subpath $p_3$ would not get augmented. In either of these two cases, or in the case when a minus edge gets empty, the odd length augmenting path of Type II from vertex $v_i$ to vertex $v_l$ would not exist.

In Lemma 15 below, we show that at any stage of Algorithm Large-cost Augmentation, an augmenting path of largest cost cannot be an even length augmenting path (either of Type I or of Type II).

**Lemma 15.** *At any stage of Algorithm Large-cost Augmentation, an even length augmenting path (either of Type I or of Type II) cannot have the largest cost.*

*Proof.** We provide a proof by induction.

**Base Case:** There are no augmenting paths of length 2. Algorithm Large-cost Augmentation initializes $x_i = 0$ for each edge $i$. Thus, each vertex is unsaturated initially. At this stage, all augmenting paths are of length 1. After one or more of these length 1 paths are augmented by the algorithm, we show that a path of length 2 cannot have the largest cost. Without loss of generality, let the path start at vertex $v_i$ and end at vertex $v_{i+2}$, and go through vertex $v_{i+1}$. Also, let $c_i > c_{i+1}$. Its existence implies that $x_{i+1} > 0$. Since vertex $v_{i+1}$ is saturated, the edge $e_{i+1}$ must have been augmented in an earlier iteration, which is a contradiction given that edge $e_i$ has higher cost and would have been augmented before edge $e_{i+1}$ gets augmented. Furthermore, edge $e_i$ would get augmented until either vertex $v_i$ or vertex $v_{i+1}$ get saturated. This results in a contradiction since an augmenting path of length 2 from vertex $v_i$ to vertex $v_{i+2}$ would not exist.

**Induction Hypothesis:** There are no even length augmenting paths of length at most $k$.

**Induction Step:** There are no even length augmenting paths of length that exceed $k$. Consider an even length augmenting path $p$ of Type I from vertex $v_i$ to vertex $v_j$ whose length exceeds $k$. Path $p$ arose because an odd length augmenting path $p_1$ of Type I from vertex $v_j$ to an intermediate vertex $v_l$ was augmented in an earlier iteration. (Note that Lemma 14 implies that path $p$ can only be of Type I.) In addition, the cost of odd length path $p_2$ from vertex $v_i$ to vertex $v_l$ exceeds the cost of path $p_1$ from vertex $v_l$ to vertex $v_j$ (otherwise the composite path $p$ cannot be
augmenting). This is a contradiction since path $p_2$ from vertex $v_i$ to vertex $v_l$ would get augmented until either $v_i$ or $v_l$ get saturated, or an even (minus) edge in path $p_2$ gets empty. In each of the above cases, the even length path $p$ would not exist.

We list below some properties of augmenting paths selected by Algorithm Large-cost Augmentation. Let $P_1, P_2, \ldots P_k$ be the sequence of augmenting paths chosen by the algorithm.

**Properties of Augmenting Paths**

1. $P_i$ is of odd length.

2. Every vertex in $P_i$, except for the two end vertices, is saturated. Odd (plus) edges of $P_i$ are incident on the unsaturated end vertices.

3. Paths $P_i$ and $P_j$, $i < j$, cannot partially overlap. However, $P_i$ can be contained in $P_j$.

4. If $P_i$ is properly contained in $P_j$ (the end vertices of $P_i$ and $P_j$ are all different), the vertices of $P_i$ are saturated when path $P_j$ is chosen for augmentation.

5. If a vertex $v$ remains unsaturated after each augmentation along the paths $P_1, P_2, \ldots, P_k$, either $v$ is not an element $P_k$ or $v$ is an end vertex of $P_k$.

We are now ready to show that the total number of augmentations is $O(n)$. Intuitively, each time an augmentation occurs, either one or both of the end vertices get saturated (and stay saturated after that), or one or more even edges (between saturated vertices) get empty. These even edges participate as minus edges in the augmentation. These edges either stay empty for the duration of the algorithm, or participate at least once as an odd edge (as a plus edge) after getting empty. However, to participate as a plus edge, the augmenting path has to get longer, which implies that at least one of the end vertices of the earlier augmenting path is saturated. In either case, we either run out of vertices, or run out of edges.

**Theorem 16.** The total number of augmentations is $O(n)$.

**Proof.** We keep a count of the number of saturated vertices in the graph in $saturated_v$. We also count the number of augmentations which empty one or more edges in an augmenting path in $empty_v$.

Note that because we use odd length augmenting paths of Type I, once a vertex gets saturated, it stays saturated for the duration of the algorithm. These vertices are intermediate vertices in an augmenting path, and continue to be saturated after the augmentation. Thus the variable $saturated_v$ is a monotonically increasing integer in the range $[0, n]$. ($n$ is the total number of vertices.)
The variable $empty_e$ is incremented each time an edge gets empty in an augmentation. Each time an empty edge in an augmenting path $p_1$ gets augmented as a plus edge in a later augmentation, the variable $empty_e$ gets decremented. But this must be preceded by at least one of the two unsaturated vertices in the earlier augmentation $p_1$ getting saturated (and staying saturated for the duration of the algorithm).

It is easy to see that the variable $empty_e$ is bounded from above by $m$ (the number of edges) because if all edges (between saturated vertices) get empty then there can be no augmenting paths. Also, the net decrease in $empty_e$ is no more than $n$ because each decrease results in an increase in $saturated_v$, whose upper bound is $n$. From this it follows that the total number of augmentations is $O(m + n) = O(2n) = O(n)$.

We are now ready to present the main result of this chapter. We will show in the next section that a max-heap data structure, as well as a balanced binary tree may be used to extract the largest cost augmenting path, as well as construct the solution in $O(\log n)$ time at each stage. We also show that there are at most $O(n)$ stages for the algorithm. This gives us an $O(n \log n)$ algorithm for the $b$-matching problem.

### 5.6 Algorithm Efficient Largest-cost Augmentation

We now show that we can extract the augmenting path of largest cost by maintaining all even and odd length augmenting paths of Type I in a max-heap data structure (however the largest cost ones will only be odd length augmenting paths of Type I). After extracting such a path, we use a balanced binary tree to determine the extent to which we can augment the path, as well as maintain the solution. Both these operations can be done in $O(\log n)$ time. We provide a description of each step of the algorithm, as well as the time it takes for each step, below.

#### 5.6.1 Description and Running Time of Algorithm Efficient Largest-cost Augmentation

Algorithm Efficient Largest-cost Augmentation starts with an initial solution $x_i = 0, i = 0, 1, \ldots, n - 1$ in Line 1. Corresponding to this solution, there are at most $n$ augmenting paths of Type I. Note that each augmenting path is between successive pairs of unsaturated vertices. We insert these paths into a max-heap. Each path is represented as an interval $(l_i, r_i)$ where $l_i$ is the left unsaturated vertex, and $r_i$ is the right unsaturated vertex (in the clockwise direction). Also associated with each interval is its cost (the key), and two pointers or indices (left and right neighbor pointers): one an index to the location of its left adjacent neighbor interval (left with respect to the actual cyclic graph) and the other an index to its right adjacent neighbor interval. We start by storing each
interval in the heap-array in the order they occur in the instance. These operations are performed in Lines 2 through 7.

We then call max-heapify to maintain the heap structure. Each time an interval is placed in the proper position in the heap-array during max-heapify, we go to its left (right) neighbor pointer, using the index to reach the left (right) adjacent interval in the heap-array. We then modify the index of its right (left) neighbor pointer, reflecting the current position of the interval in the heap-array. The above procedure is called Build-Max-Heap (Line 8), and it takes at most $O(n)$ time.

The data stored in the balanced binary tree data structure is initialized in Line 9. Details are provided in Section 5.7. As long as an augmenting path exists, Algorithm Efficient Large-cost Augmentation picks the augmenting path of largest cost from the max-heap. This is performed in Line 10 and takes $O(1)$ time.

The cost of the interval with the maximum cost is checked in Line 11. The algorithm continues executing as long as this cost is $> 0$ (and terminates if the cost equals 0). The interval corresponding to the largest augmenting path, max_cost_int, is given as input to Algorithm Augment-Interval in Line 12. Algorithm Augment-Interval keeps track of the saturated and unsaturated vertices, as well as the current solution. Algorithm Augment-Interval returns the bottleneck, an integer corresponding to the extent to which the interval max_cost_int can be augmented, and a vector of three elements, denoting which of the following (not necessarily exclusive) events occur on augmentation: left vertex gets saturated, right vertex gets saturated, one or more edges in the even position gets empty. Algorithm Augment-Interval runs in $O(\log n)$ time each time it is called. This is discussed in Section 5.7.

Lines 13 through 33 use the output of Algorithm Augment-Interval to modify the heap which stores all augmenting paths in the graph. If the output of Algorithm Augment-Interval denotes that the left vertex is saturated after augmentation, then the interval $(l_i, r_i)$ as well as its left adjacent interval $(l_{i-1}, r_{i-1})$ (note that $l_i = r_{i-1}$) are removed from the heap-array, and a new interval $(l_{i-1}, r_i)$ is inserted into the heap-array. Note that the above operation corresponds to a merging of the two intervals. Similarly, if the output of Algorithm Augment-Interval denotes that the right vertex is saturated after augmentation, then the interval $(l_i, r_i)$ as well as its right adjacent interval $(l_{i+1}, r_{i+1})$ (note that $r_i = l_{i+1}$) are removed from the heap-array, and a new interval $(l_i, r_{i+1})$ is inserted into the heap-array. If both the left and the right vertices are saturated, then interval $(l_i, r_i)$ as well its left and right adjacent interval are removed from the heap-array, and a new interval $(l_{i-1}, r_{i+1})$ corresponding to a merging of all three intervals is inserted. Finally, if one or more edges get empty, then the newly inserted interval is removed from the heap-array and inserted into the bottom of the heap-array. The left and right neighbor pointers of its adjacent intervals, as well as its own left and right neighbor pointers are also suitably updated to reflect its new position. All this takes $O(\log n)$ time.

The pseudocode for Algorithm Efficient Large-cost Augmentation is given on the next page.
Algorithm 13: Algorithm Efficient Large-cost Augmentation

1. Initialize $x_i = 0$ for $i = 0, 1, \ldots, n - 1$  
   — The solution is initialized.

2. for $i = 0$ to $n$ do

3.   heap-array[i].int = $(v_{i-1}, v_i)$
4.   heap-array[i].cost = $c_i$
5.   heap-array[i].left = $(v_{i-2}, v_{i-1})$
6.   heap-array[i].right = $(v_i, v_{i+1})$

   ▷ heap-array is initialized.

7. Build-Max-Heap(heap-array)

8.   — heap-array is sorted in decreasing order of cost of intervals. When an interval is
   moved during the sorting, the right and left pointers of its left and right intervals are changed
   to reflect its new position.

9. Initialize balanced binary tree   ▷ Algorithm Augment-Interval uses balanced binary tree.

10. max_cost_int = max(heap-array)

11. while max_cost_int.cost > 0 do

12.   (bottleneck, left_vertex, right_vertex, edge_empty) = Augment-Interval(max_cost_int)

13.   ▷ Algorithm Augment-Interval augments the
   path by the maximum extent possible, and returns the bottleneck, whether the left end vertex
   and or the right end vertex get saturated, and or one or more minus edges get empty.

14. if left_vertex = true then

15.   left_int = Heap-Extract(max_cost_int(left))
16.   int = merge(left_int, max_cost_int)

17.   — The left interval of max_cost_int is removed from heap-array and merged with interval
   max_cost_int.

18. if length of left_int is even then

19.   cost = left_int.cost + max_cost_int.cost
20. else
21.   cost = left_int.cost - max_cost_int.cost

22.   — Cost of new interval is computed.
23. else
24.   int = max_cost_int;

25. if right_vertex = true then

26.   right_int = Heap-Extract(max_cost_int(right))
27.   int = merge(right_int, int)

28.   — The right interval of max_cost_int is removed from heap-array and merged with interval
   max_cost_int.
29.   cost = int.cost - right_int.cost

30.   — Cost of new interval is computed.
31. if edge_empty = true then

32.   interval int is placed at the bottom of heap-array
33. else
34.   Heap-Insert(int)

35.   — Otherwise it is inserted in sorted order. In both cases, the left and right pointers of its left
   and right intervals are adjusted to reflect its current position.

36. max_cost_int = max(heap-array)

65
5.6.2 Running Time of Algorithm Efficient Large-cost Augmentation

We now show that the running time of Algorithm Efficient Large-cost Augmentation is \( O(n \log n) \).

**Theorem 17.** Algorithm Efficient Large-cost Augmentation runs in time \( O(n \log n) \).

**Proof.** The initialization for Algorithm Efficient Large-cost Augmentation takes time at most \( O(n \log n) \), determined by Build-Max-Heap in Line 8. Initializing the balanced binary tree for Algorithm Augment-Interval takes \( O(n) \) time (Lemma 18). Algorithm Efficient Large-cost Augmentation executes Line 11 through Line 35 repeatedly until a maximum cost augmenting path has cost 0 (until there is no augmenting path). From Theorem 16, it follows that there are at most \( O(n) \) augmentations. We will now determine the time for each statement in this loop which runs \( O(n) \) times. Line 10 takes \( O(1) \) time. Line 12 calls Algorithm Augment-Interval which takes \( O(\log n) \) time (Lemma 21). Lines 13 through 28 perform Heap-Extract which takes \( O(\log n) \) time. Lines 29 through 33 perform Heap-Insert which takes \( O(\log n) \) time. Thus, Algorithm Efficient Large-cost Augmentation runs in \( O(n \log n) \) time.

We next provide the balanced binary tree data structure used to update and store the solution, as well as a description of how Algorithm Augment-Interval updates the data structure.

5.7 Details of Algorithm Augment-Interval

In the next subsection we provide a detailed description of Algorithm Augment-Interval. Recall that the input to Algorithm Augment-Interval is an interval, and its output is the vector (bottleneck, left_vertex, right_vertex, edge_empty).

To compute the output, we need to know the amount we can add to the first and last edges before either the first vertex or the last vertex or both vertices get saturated. We also need to know the amount we can subtract from the even edges before one or more of these even edges get empty. The minimum of these two amounts determines the amount by which we can augment the path. A naive implementation can represent the graph as an adjacency list, with the capacity of each vertex stored with the vertex, and the current value assigned to each edge in the graph stored along with the edge. An augmenting path can be extracted by traversing a path from each unsaturated vertex. The amount that can be added to the first and last edges before either end vertex (or both) get saturated can be computed in \( O(1) \) time. The amount that can be subtracted from the even edges (before one or more of these get empty) can be computed in \( O(n) \) time simply by taking the minimum of the solution values assigned to each of these edges.

The balanced binary tree data structure allows us to compute the amount we can add to (subtract from) the odd (even) edges in an augmenting path in \( O(\log n) \) time. The interval corresponding to the augmenting path is divided into subranges (or subintervals), and the amount that can be
subtracted from the even edges is computed as the minimum that can be subtracted from the even edges in each subrange. The number of these subranges is \(O(\log n)\), and therefore computing the minimum of these subranges can be computed in \(O(\log n)\) time. We provide the details next.

### 5.7.1 Detailed Description

We construct a balanced binary tree \(T\) whose leaf nodes represent the edges. Each internal node \(u\) represents an interval \([v_i, v_j]\) and the edges spanning this interval are \((v_i, v_{i+1}), (v_{i+1}, v_{i+2}), \ldots, (v_{j-1}, v_j)\). We call the path (edges) the spanning path (edges) of node \(u\) or of \(T_u\), the subtree rooted at \(u\). We consider the edges of the cycle in clockwise order.

Consider augmenting path \(P = [v_k, v_{k'}]\). If \(k < k'\), the edges in the path are \(\{e_k, e_{k+1}, \ldots, e_{k'-1}\}\). If \(k' < k\), the edges in the path are \(\{e_k, e_{k+1}, \ldots, e_{n-1}, e_0, e_1, \ldots, e_{k'-1}\}\). Let \(\pi(e_k)\) and \(\pi(e_{k'-1})\) denote the paths from the root of tree \(T\) to the leaf nodes representing \(e_k\) and \(e_{k'-1}\) respectively. A subtree \(T_u\) is called a canonical subtree of \(P\) if and only if \(P\) contains the spanning path of \(u\), but does not contain the spanning path of its parent node. Thus the paths \(\pi(e_k)\) and \(\pi(e_{k'-1})\) determine the set of canonical subtrees of \(T\) whose spanning paths, when concatenated, comprise the edges of augmenting path \(P\). Since \(T\) is a balanced binary tree with \(n\) leaf nodes, the number of canonical subtrees of any path from the root of \(T\) to a leaf is at most \(O(\log n)\). For more details on canonical subsets in the context of range trees, see [36].

In order to determine the bottlenecks of the edges after a sequence of augmentations, every node \(u\) stores a 3-tuple \((m_{even}(u), m_{odd}(u), \text{offset}(u))\). \(m_{even}(u)\) is the bottleneck of its spanning edges when the leftmost spanning edge of \(u\) is an even edge of an augmenting path, \(m_{odd}(u)\) is the bottleneck when the leftmost spanning edge of \(u\) is an odd edge of an augmenting path, and \(\text{offset}(u)\) denotes the solution, the net amount we have added to the spanning edges of \(u\) that are incident on unsaturated vertices, due to augmentations along augmenting paths which have \(T_u\) as a canonical subtree (Note that \(T_u\) will be a canonical subtree only of longer augmenting paths, that occur later in Algorithm Efficient Large-cost Augmentation).

During the augmentation of paths, the following invariance is maintained at every internal node \(u\): \(m_{even}(u)\) and \(m_{odd}(u)\) respectively denote the bottleneck of even and odd edges of \(T_u\) from those augmenting paths either contained in \(T_u\) or have \(T_u\) as one of the canonical subtrees. Algorithm Augment-Interval initializes the 3-tuple stored at each leaf node \(u\) by assigning a value 0 for \(m_{even}(u)\) and a value 0 for \(\text{offset}(u)\) (corresponding to a path of length 1). Note that this offset value corresponds to the initial solution value \(x_e = 0\) assigned to each edge \(e\) in the graph. The initial value assigned to \(m_{odd}(u)\) for a leaf node is \(\infty\) because this unsaturated edge can participate as an odd edge only. The offset value of each internal node is set to 0. The 3-tuples stored in the internal nodes of \(T\) are now updated using the initialized values for the 3-tuples at the leaf nodes.

We now describe how the 3-tuples stored at each internal node \(u\) gets updated using the values of the 3-tuples stored at its children. Let \(u_l\) \((u_r)\) denote the left (respectively right) child of \(u\). Let
$e_l$ ($e_r$) be the first edge of the spanning path of $T_{ul}$ ($T_{ur}$). The 3-tuple stored at $u$ is updated using Procedure Update Parent on the next page:

**Procedure Update Parent**

$$m_{odd}(u) = \begin{cases} \min\{m_{odd}(u_l), m_{odd}(u_r)\} & \text{if } e_r \text{ is an odd edge of } T_u \\ \min\{m_{odd}(u_l), m_{even}(u_r)\} & \text{if } e_r \text{ is an even edge of } T_u \end{cases}$$

$$m_{even}(u) = \begin{cases} \min\{m_{even}(u_l), m_{odd}(u_r)\} & \text{if } e_r \text{ is an odd edge of } T_u \\ \min\{m_{even}(u_l), m_{even}(u_r)\} & \text{if } e_r \text{ is an even edge of } T_u \end{cases}$$

Note that the left child (subtree) can, in general, have an odd number of edges. If the left child (subtree) does have an odd number of edges, then $e_r$ can be an odd (even) edge of $T_u$ while $e_l$ is an even (odd) edge of $T_u$.

Clearly, the 3-tuples stored at each leaf node of tree $T$ can be computed in constant time. Using these 3-tuples, the 3-tuples stored in the remaining nodes of the balanced binary tree $T$ can be computed using Procedure Update Parent in linear time (Lemma 18).

**Lemma 18.** The 3-tuples stored in the nodes of the balanced binary tree $T$ can be initialized in $O(n)$ time.

We now look at what happens when a valid solution, say $\delta$, for an augmenting path $P$ is added to the spanning edges of $u$. Note that $T_u$, the subtree rooted at node $u$ is a canonical subtree of path $P$. Recall that $e_l$ is the first edge of the spanning path of $u$. The 3-tuple stored at node $u$ is updated using Procedure Augment Path below:

**Procedure Augment Path**

$$m_{odd}(u) = \begin{cases} m_{odd}(u) - \delta & \text{if } e_l \text{ is an odd edge with respect to } P \\ m_{odd}(u) + \delta & \text{if } e_l \text{ is an even edge with respect to } P \end{cases}$$

$$m_{even}(u) = \begin{cases} m_{even}(u) + \delta & \text{if } e_l \text{ is an odd edge with respect to } P \\ m_{even}(u) - \delta & \text{if } e_l \text{ is an even edge with respect to } P \end{cases}$$
offset(u) = \begin{cases} 
\text{offset}(u) + \delta & \text{if an edge in the spanning path of } u \text{ is incident on an unsaturated vertex in } P \\
\text{offset}(u) & \text{otherwise}
\end{cases}

Note that we need to maintain the following invariance at each node \( u \) at all times: \( m_{\text{even}}(u) \) and \( m_{\text{odd}}(u) \) respectively denote the bottleneck when the leftmost spanning edge of \( u \) is an even or an odd edge of an augmenting path. This bottleneck is a result of the augmentation of those paths that have \( T_u \) as one of the canonical subtrees. Furthermore, \( \text{offset}(u) \) denotes the net amount we have added to a spanning edge of \( u \) incident on an unsaturated vertex, due to augmentations along augmenting paths which have \( T_u \) as a canonical subtree. If \( T_u \) doesn’t contain any unsaturated vertex, \( \text{offset}(u) \) is never used. Therefore, after modifying the 3-tuple stored at \( u \), we still need to update the 3-tuples stored at the nodes lying on the path from the root of tree \( T \) to node \( u \). Let \( \pi(u) \) be the path from the root to \( u \). Let \( p_i \) be the parent node of \( u_i \) and \( p_{i+1}, i = 1, 2, ..., t \) and \( u = p_{t+1} \). We update the 3-tuples stored at nodes \( p_i, i = t, ..., 1 \) from the bottom up using Procedure Update Parent. Lemma 19 thus follows.

**Lemma 19.** For a valid augmentation \( \delta \), the 3-tuples stored at node \( u \), and at its ancestor nodes all the way to the root, can be updated in \( O(\log n) \) time.

Consider a leaf node \( u \) whose spanning edge \( e \) has one of its end vertices unsaturated. Consider the path \( \pi(e) \) from the root of \( T \) to leaf \( u \).

**Lemma 20.** The value assigned by Algorithm Efficient Large-cost Augmentation to edge \( e \) at any given time during the execution of the algorithm is \( \sum_{w \in \pi(e)} \text{offset}(w) \), and this can be computed in \( O(\log n) \) time.

**Proof.** Since \( e \) is incident on an unsaturated vertex, any augmenting path containing \( e \) that is chosen by the algorithm either starts at \( e \) or ends at \( e \). Furthermore, each of these augmenting paths is of odd length, and edge \( e \) is always a plus edge in these paths. 

Next, we explain how we compute the maximum amount by which an augmenting path \( P \) may be augmented. Consider what happens when Algorithm Augment-Interval is called with an augmenting path \( P \) from \( v_k \) to \( v_{k'} \), \( k < k' \). Let \( T_{z_i}, i = 1, 2, ..., t \) be the canonical subtrees of \( P \) where \( t \) is \( O(\log n) \). We need to find the minimum value assigned to the even edges of \( P \). We denote this value by \( t_{e}(P) \). Let \( O \) (alternatively \( E \)) be the subset of the roots of the canonical subtrees of \( P \) whose leftmost edge is an odd (alternatively even) edge of path \( P \). \( t_{e}(P) \) is given by the following expression: \( t_{e}(P) = \min\{\min_{u \in O} m_{\text{odd}}(u), \min_{u \in E} m_{\text{even}}(u)\} \).
We also need to find the amount by which the first edge and the last edge of path \( P \) may be augmented. These are denoted by \( t_f(P) \) and \( t_l(P) \) respectively. Since both the edges \( f \) and \( l \) are incident on unsaturated vertices, \( t_f(P) \) and \( t_l(P) \) can be computed using Lemma 20. The unsaturated vertex \( v_k \) is incident to edges \( e_{k-1} \) and \( e_k \). The values assigned to these edges, \( x_{e_{k-1}} \) and \( x_{e_k} \), can be computed in \( O(\log n) \) time, using Lemma 20. Similarly \( t_l(P) \) can be computed in \( O(\log n) \) time. Let \( t(P) = \min\{t_e(P), t_f(P), t_l(P)\} \). Our updating step must decrease the value assigned to each of the even edges by \( t(P) \) and increase the value assigned to each of the odd edges by \( t(P) \). We use Procedure Augment Path to update the 3-tuples stored at each root node \( z_i \) of canonical subtree \( T_{z_i} \).

We also update, for each \( i, i = 1, 2, \ldots, t \), the 3-tuples stored at the nodes along the path from the root node \( z_i \) of \( T_{z_i} \) to the root node of the balanced binary tree \( T \). The parent nodes of these subtrees either lie on the path from the root of \( T \) to \( v_k \) or on the path from the root of \( T \) to \( v_{k'} \). Using Procedure Update Parent, the 3-tuples stored at all these nodes can be updated in \( O(\log n) \) time.

An almost identical approach can be used to handle the case when the augmenting path is from \( v_k \) to \( v_{k'} \) where \( k > k' \).

**Lemma 21.** For each augmenting path, Algorithm Augment-Interval takes \( O(\log n) \) time.

**Proof.** Since any augmenting path \([v_i, v_j]\) can be represented by the union of \( O(\log n) \) canonical subtrees, \( t_e([v_i, v_j]) \) can be obtained in \( O(\log n) \) by taking the minimum of \( O(\log n) \) values corresponding to each canonical subtree. To compute \( t_f([v_i, v_j]) \) and \( t_l([v_i, v_j]) \), we traverse \( O(\log n) \) nodes from a leaf to the root of the tree. Thus, the entire process takes \( O(\log n) \) time.

The solution \( x(e) \) assigned to each edge \( e \in E \) can be reconstructed from the 3-tuples stored in the binary search tree. We outline below a procedure to accomplish this in \( O(n \log n) \) time.

### 5.7.2 Constructing the Solution

When Algorithm Efficient Largest-cost Augmentation terminates, the values assigned to each edge can be computed easily. To keep track of the saturated vertices in the graph, every time a vertex is saturated (output of Algorithm Augment-Interval), an array with a boolean entry for each vertex is updated. There are two cases:

**Case I:** There are some unsaturated vertices in the graph. For each unsaturated vertex, we traverse the path from one of its incident edges to the root of the balanced binary tree. The value assigned to the edge is given by the sum of the offset values of the nodes on this path. The value assigned to the other incident edge can also be computed in the same way. Once the values assigned to the edges incident on unsaturated vertices is known, the values assigned to edges incident on saturated
vertices can be computed by observing that for each saturated vertex, the sum of the values assigned to the edges incident on the vertex equals the capacity of the vertex.

**Case II.** There is no unsaturated vertex in the graph. In this case we need the final augmenting path (before Algorithm Efficient Largest-cost Augmentation terminated), as well as the bottleneck returned by Algorithm Augment-Interval for the final augmenting path. By subtracting the bottleneck from the end edges of the augmenting path, we can construct the solution before the last augmentation took place. Using the method outlined in Case I, we can compute the value assigned to each edge in the graph. The optimal solution can now be computed by adding (subtracting) the bottleneck to (from) each odd (even) edge of the augmenting path.

The pseudocode for Algorithm Augment-Interval, as well as an example to illustrate its details are provided below.

```
Algorithm 14 Algorithm Augment-Interval
1: Input: A path \([v_i, v_j]\]
2: \(left\_vertex, right\_vertex, edge\_empty = false\)
3: Find all the canonical subtrees \(T_1, T_2, ..., T_m\) of path \([v_i, v_j]\)
4: for \(i \leftarrow 1\) to \(m\) do
5:   check if the leftmost edge in \(T_i\) is located in even or odd position with respect to path \([v_i, v_j]\)
6:   if the leftmost edge in \(T_i\) is located in even position then
7:     \(minimum\_even\_edge[i] \leftarrow m_{even}(i)\)
8:   else
9:     \(minimum\_even\_edge[i] \leftarrow m_{odd}(i)\)
10: \(t_e([v_i, v_j]) \leftarrow \min(minimum\_even\_edge)\)
11: \(t_f([v_i, v_j]) \leftarrow \text{maximum amount that can be added to the leftmost edge}\)
12:  \(\triangleright\) We start from the leaf nodes corresponding to \((v_{i-1}, v_i)\) and \((v_i, v_{i+1})\), traverse the tree to the root and update the amounts according to the offsets stored on the nodes along the paths.
13: \(t_l([v_i, v_j]) \leftarrow \text{maximum amount that can be added to the rightmost edge}\)
14: \(\triangleright\) Similar to \(t_f\) procedure.
15: \(bottleneck \leftarrow \min(t_e([v_i, v_j]), t_f([v_i, v_j]), t_l([v_i, v_j]))\)
16: Update for each \(i, i = 1, 2, ..., m\), the 3-tuples stored at the nodes along the path from the root node \(z_i\) of \(T_{z_i}\) to the root node of the balanced binary tree \(T\) using Procedure Update Parent
17:
18: if \(bottleneck = t_e([v_i, v_j])\) then
19:   \(edge\_empty = true\)
20: if \(bottleneck = t_f([v_i, v_j])\) then
21:   \(left\_vertex = true\)
22: if \(bottleneck = t_l([v_i, v_j])\) then
23:   \(right\_vertex = true\)
24: Output: \(\{bottleneck, left\_vertex, right\_vertex, edge\_empty\}\)
```
**Example 1.** We provide an example for Algorithm Augment-Interval when \( n = 7 \). We show how the values in the leaves and internal nodes change after each augmentation. We also provide a snapshot of the cyclic graph after each augmentation. The vector of capacities for the vertices is \( \{v_0, v_1, v_2, v_3, v_4, v_5, v_6\} = \{2, 5, 3, 9, 10, 14, 5\} \). Figure 5.5 shows the initial values of the 3-tuples stored in each node of the balanced binary tree. The first augmenting path is \( \{v_4, v_5\} \) and the leaf corresponding to edge \( e_4 \) is the only canonical subtree of this augmenting path. Since edge \( e_4 \) can only be used as an odd edge, \( m_{\text{odd}}(e_4) = \infty \), and the bottleneck for this path is 10, the minimum of \( t_f(\{v_4, v_5\}) = 10 \) and \( t_l(\{v_4, v_5\}) = 14 \). In accordance with Procedure Augment Path in Section 5.7, we add 10 to the current offset value and the \( m_{\text{even}} \) value in the leaf node corresponding to \( e_4 \). All the nodes along the path from the leaf to the root node of the entire tree are also updated in accordance with Procedure Update Parent in Section 5.7 (refer to Figure 5.6). \( \{v_6, v_0\} \), \( \{v_5, v_0\} \), and \( \{v_1, v_2\} \) are the next 3 augmenting paths. Finding their bottlenecks and updating the corresponding 3-tuples is similar. Figures 5.7, 5.8, 5.9 show the tree after each augmentation.

![Figure 5.5: Initial state](image1.png)

![Figure 5.6: Path \( e_4 = \{v_4, v_5\} \)](image2.png)
Figure 5.7: Path $e_6 = \{v_6, v_0\}$

Figure 5.8: Path $e_5 = \{v_5, v_6\}$

Figure 5.9: Path $e_1 = \{v_1, v_2\}$
Chapter 6

Conclusions

In this chapter, we lay out our results and the future scope for the problems addressed in this dissertation.

6.1 Winner Determination Problem and Heuristics

In Chapter 2, we compute the average solution value and the domination number for the winner determination problem under the Chamberlin-Courant system. We show that the 2-Opt heuristic can obtain a solution no worse than the average in polynomial time, and hence achieves the domination ratio of $\frac{1}{|C|}$ ($|C|$ is the number of candidates).

We also run some experiments and test the theoretical results. It would be interesting to consider the 2-Opt heuristic for the weighted version of Chamberlin-Courant system, which is introduced in Chapter 1 as the Monroe system, and obtain analogous results.

6.2 Linear Ordering Problem

The Linear Ordering Problem is a classic optimization problem with broad applications. Solving the problem optimally for the larger hard instances has proved difficult. In Chapter 3, we report our experience with using a MIP heuristic for the problem. Our heuristic generates a starting feasible solution based on the LP relaxation of the IP formulation for the Linear Ordering Problem. For each starting solution, a neighborhood is defined, again based on the LP solution to the problem. A MIP solver is then used to obtain the optimal solution among all the solutions in the neighborhood. Preliminary results indicate that this approach is promising. The solutions obtained using this heuristic are substantially closer to the optimal than the solutions obtained using the MIP solver to solve the entire problem.

In Chapter 4, we show that the linear programming formulation for the linear ordering problem can be solved by a combinatorial algorithm. Both optimal and near-optimal methods for solving the
linear ordering problem can benefit from an efficient algorithm for this linear program. For future research, it would be interesting to design a combinatorial algorithm based on the augmenting structures obtained in this chapter.

6.3 Cyclical Shift Scheduling

We provide an $O(n \log n)$ algorithm for the $b$-matching problem on a cycle. The solution to this problem may be used to solve the $(n-2, n)$ cyclic scheduling problem in $O(n \log n)(\log nb_{\max})$ time. An interesting open problem would be to provide a strongly polynomial algorithm for the $(n-3, n)$ cyclic scheduling problem, or more generally, the $(k, n)$ cyclic scheduling problem efficiently. One way to accomplish this is to adapt Megiddo’s parametric search [70] for the $b$-matching problem on a hypergraph. The approach given in this research can be used to solve the $b$-matching problem on any graph as long as the augmenting paths that arise can be stored efficiently, as well as the values assigned to the edges modified efficiently. Thus, it is not clear if this approach can be readily adapted to solve the $b$-matching problem for a general planar graph.
Bibliography


