Algorithms for Solving

Variations of the Traveling Salesman Problem

by

Pouya Baniasadi
B.Math, B.Sc. (Hons)

Thesis submitted for the degree of
Doctor of Philosophy
College of Science and Engineering
Flinders University

Oct 2019
To my best friend Andrew Childs (1986 - 2017),
for the beautiful memories that will always be with me.
Contents

Dedication i

Table of Contents ii

List of Figures v

List of Tables vii

Summary x

Declaration xi

Acknowledgements xiii

1 Introduction 1

1.1 The Traveling Salesman Problem . . . . . . . . . . . . . . . . . . 1

1.2 Mathematical Preliminaries . . . . . . . . . . . . . . . . . . . . 2

1.3 Solving the Traveling Salesman Problems . . . . . . . . . . . . . 8

1.3.1 Lower Bounds for TSP . . . . . . . . . . . . . . . . . . . 8

1.3.2 Upper Bounds for TSP: Tour Improvement Heuristics . . . 10

1.4 Variations of the Traveling Salesman Problem . . . . . . . . . . 12

2 A Novel Tour Improvement Approach for The Hamiltonian Cycle 16

Problem

2.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
2.1.1 Notation .................................................. 18
2.1.2 Highlights of the Snakes and Ladders Heuristics ............ 19
2.2 Tour Improvement and the Blind Improvement Space .......... 23
2.3 Design Fundamentals of a BIS based Algorithm ................. 26
2.4 Principle 1: Constructing a Blind Improvement Space ........... 27
2.5 Principle 2: Restricting the Use of Opening Transformations .... 34
2.6 Principle 3: Prioritizing Transformations with Lower Computa-
tional Cost .................................................. 38
2.7 The Basic Algorithm ......................................... 40
2.7.1 Software ................................................. 43

3 Benchmarking Traveling Salesman Problem Algorithms by Diffi-
cult Sparse Instances ........................................ 44
3.1 Introduction ................................................. 45
3.1.1 TSP Benchmarks ......................................... 48
3.1.2 Properties that Contribute to the Difficulty of Instances .... 50
3.1.3 Algorithms to be Tested .................................. 51
3.2 HCP Instances in the Literature ................................ 53
3.2.1 Generalized Peterson Graphs ................................ 53
3.2.2 Sheehan Graphs .......................................... 55
3.2.3 Modified Flower Snarks .................................. 55
3.2.4 Fleischner Graphs ........................................ 56
3.3 Randomly Generated Instances .................................. 57
3.3.1 Randomly Generated HCP Instances ......................... 58
3.3.2 Modifying Random Instances ............................... 58
3.4 HCP Instances Corresponding to Other $\mathcal{NP}$-complete Problems .......................... 63
3.4.1 Converting Computationally Difficult Problems to HCP .... 63
3.5 Discussion ..................................................... 65

4 The Sparse Traveling Salesman Problem ........................... 67
4.1 Introduction ................................................... 67
List of Figures

1.1 An example of an undirected 10 node graph .......................... 3

2.1 $\gamma(y, x, a)(T)$ transformation. ........................................ 19
2.2 $\varpi$ transformation $\varpi[y, x, a, b, c](T)$. .......................... 20
2.3 Example of a tour of a graph which can not be improved by a
2-exchange transformation. .................................................... 30
2.4 Floating $\gamma$ transformation $\gamma[y, x, a](T)$. ......................... 30
2.5 Opening $\psi$ transformation $\psi[y, x, a, b](T)$. ....................... 31
2.6 Case 1 in the proof of Proposition 2.4.6 ................................. 32
2.7 Case 2 in the proof of Proposition 2.4.6 .................................. 33
2.8 Opening $\varpi$ transformation $\varpi[y, x, a, b, c](T)$. .................. 35
2.9 Case 1 in the proof of Proposition 2.5.2 ................................. 36
2.10 Case 2 in the proof of Proposition 2.5.2 ................................. 37
2.11 Case 3 in the proof of Proposition 2.5.2 ................................. 37

4.1 Example of a sequential transformation. ................................. 75
4.2 Example of a non-sequential transformation. ............................ 76
4.3 In-out-subgraph of $C_i$ for $n = 5$. ..................................... 89
4.4 Examples of graphs $\Gamma$ and $G$ for $n = 3$. ........................... 92
4.5 Example of of construction of the overlap graph $G$ for the given reads.101
4.6 Example of tour $T$ of $G$ mapped to the read sequence of $T$. ........ 103

5.1 The linkages between GTSP, CTSP and CGTSP. ....................... 113
5.2  A feasible tour of the CGTSP with five clusters. .......................... 118
5.3  TSP tour of $G'$ made by transforming the CGTSP graph $\Gamma$ of Figure
     5.2. ......................................................................................... 127
5.4  The rectangular layout of a robotic ASRS with random and volume-
     based storage policies. .......................................................... 140
5.5  An example of a CGTSP route for a drone-assisted PDS in Mel-
     bourne central business district. ............................................. 143
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>The four situation where a test instance is counted as unsolved</td>
<td>52</td>
</tr>
<tr>
<td>3.2</td>
<td>Three types of Generalized Peterson graphs $GP(n/2, k)$ and their respective number of Hamiltonian cycles</td>
<td>53</td>
</tr>
<tr>
<td>3.3</td>
<td>Performance of the four algorithms on generalized Peterson graphs</td>
<td>54</td>
</tr>
<tr>
<td>3.4</td>
<td>Performance of the four algorithms on Sheehan Graphs</td>
<td>55</td>
</tr>
<tr>
<td>3.5</td>
<td>Performance of the four algorithms on modified Flower Snarks</td>
<td>56</td>
</tr>
<tr>
<td>3.6</td>
<td>Performance of the four algorithms on Fleischner graphs</td>
<td>57</td>
</tr>
<tr>
<td>3.7</td>
<td>Performance of the four algorithms on randomly generated HCP instances</td>
<td>59</td>
</tr>
<tr>
<td>3.8</td>
<td>Performance of LKH on modified randomly generated HCP instances</td>
<td>62</td>
</tr>
<tr>
<td>3.9</td>
<td>Performance of CLK on modified randomly generated HCP instances</td>
<td>62</td>
</tr>
<tr>
<td>3.10</td>
<td>Performance of Concorde on modified randomly generated HCP instances</td>
<td>62</td>
</tr>
<tr>
<td>3.11</td>
<td>Performance of SLH2 on modified randomly generated HCP instances</td>
<td>62</td>
</tr>
<tr>
<td>3.12</td>
<td>Performance of the four algorithms on converted computationally difficult problems to HCP</td>
<td>65</td>
</tr>
<tr>
<td>4.1</td>
<td>Average performance of LKH on 1000 random regular TSP graphs with degree 3</td>
<td>77</td>
</tr>
</tbody>
</table>
4.2 Comparative performance of SLH and two altered settings of LKH on random regular graphs .......................................................... 79
4.3 Performance of Concorde-SLH and sLKH-SLH on Chapter 3 instances .......................................................... 83
4.4 Orientation edges of $E_1$ for each $i = 1, 2, \ldots, n$ ........................................ 99
4.5 Overlap edges for the four possible ways that $R_i$ and $R_j$ may overlap 100
4.6 Edges of $E_0$ for $i = 1, 2, \ldots, n$ .......................................................... 100
4.7 LKH parameters for solving the simulated DNA-sequencing examples of Table 4.8 .......................................................... 109
4.8 Performance of sLKH on simulated instances of TSP overlap compared to a greedy algorithm .......................................................... 110
5.1 A matrix of weights of edges for a CGTSP .......................................................... 119
5.2 List of notations used in Chapter 5 .......................................................... 120
5.3 Recommended LKH parameters for solving the transformed CGTSP 133
5.4 Comparative performance of the IP versus the TSP formulation of CGTSP for exact solutions .......................................................... 134
5.5 Comparative performance of the IP versus TSP formulation of CGTSP for heuristic solutions .......................................................... 135
5.6 Performance of LKH on large scale CGTSP instances .......................................................... 136
5.7 Comparison of random and alternative warehouse arrangements .......................................................... 142
5.8 Drone-assisted PDS on GTSPLIB instances with different number of drones .......................................................... 146
A.1 Performance of TSP algorithms on the generated TDTSP instances 154
Summary

The Traveling Salesman Problem (TSP) is widely considered to be one of the most important problems in optimization and computer science. The reasons behind the significance of TSP are both its theoretical connections to complexity theory and algorithmic optimization, and its potential applications in various fields such as logistics and genetics. In this thesis, I consider three important variations of TSP from the standpoint of algorithmic solutions and applications. These variations are the Hamiltonian Cycle Problem (HCP), the Sparse Traveling Salesman Problem (STSP) and the Clustered Generalized Traveling Salesman Problem (CGTSP).

Given a graph, the HCP is the problem of finding a simple cycle that includes all nodes of the graph, and it is equivalent to the TSP where the weights are binary values. Despite its deceptively simple definition, there is no known efficient algorithm for solving the HCP as it belongs to the important class of $\mathcal{NP}$-complete problems. As a special and difficult subset of the TSP, the study of HCP is a potentially rewarding direction of research for understanding complexities involved in the TSP itself. In Chapter 2, I analyze and further develop a recently introduced approach for solving the HCP called the Snakes and Ladders Heuristic (SLH). SLH is demonstrated to be on par with state-of-the-art approaches for solving HCP, and in some instances, it outperforms other approaches. An abstract discussion of the algorithmic principles at play in SLH is included, and accordingly, an improved
algorithmic approach for HCP is proposed.

Sparse Traveling Salesman Problem (STSP) is a slight variation of the TSP on sparse instances. The primary motivation for considering this special case of TSP is its algorithmic significance. Most successful algorithmic approaches for solving TSP involves the initial sparsification of the instance, suggesting that much of the computational complexity of TSP comes from STSP. In Chapter 3, I discuss the creation of meaningful benchmark instances of STSP that provide insightful information about the performance of a TSP algorithm. Using these instances, I report on a computational study, involving over eight years of CPU time, and examine some of the state-of-the-art TSP algorithms. Next, I use the insights gained from the computational study to identify a set of effective algorithmic approaches for solving STSP in Chapter 4. I implement and combine these approaches to produce effective hybrid algorithms for the STSP. Finally, I explore the applications of STSP algorithms by formulating two well-known problems in terms of an STSP. These two problems are the Time-Dependant Traveling Salesman Problem and the DNA-assembly problem.

In Chapter 5, I describe an extension of the TSP called CGTSP. From the perspective of real-world applications, a CGTSP model provides much more flexibility for accommodating a wider range of assumptions compared to a classical TSP model. I develop a transformation method for solving CGTSP and demonstrate substantial improvements to the existing solutions methods for CGTSP. Finally, I then illustrate the potential applications of the CGTSP in two modern problems in logistics, namely the robotic automated storage and retrieval system and the drone-assisted delivery systems. Finally, I report a comprehensive computational study on the effectiveness of the transformation method.
Declaration

This dissertation describes work carried out at Flinders University of South Australia under the principal supervision of A/Prof. Vladimir Ejov and co-supervision of Prof. Jerzy Filar and Dr. Michael Haythorpe. I certify that this thesis does not incorporate without acknowledgment any material previously submitted for a degree or diploma in any university; and that to the best of my knowledge and belief it does not contain any material previously published or written by another person except where due reference is made in the text.

Pouya Baniasadi
Acknowledgements

First, I would like to express my deep gratitude to my supervisory team. I thank A/Prof. Vladimir Ejov, my principal supervisor, for his guidance and insight throughout my candidature. I am very appreciative for my exposure to the learning opportunities and fascinating research topics while working with Vlad and his team. His patience, kindness and passion has made my research project a memorable and enjoyable journey.

During my first year as an undergraduate student, Prof. Jerzy Filar offered an incredible opportunity to work with his combinatorial optimisation research team at University of South Australia. That rare opportunity is the reason that I chose to study mathematics and continue to undertake a Ph.D. Ever since, Jerzy has been an incredibly generous mentor, both in academic and personal capacity.

The support and the intellectual guidance of Dr. Michael Haythorpe was tremendously constructive. For many years, I have seen him as a model of having a strong work ethic, teaching with passion, and being a well rounded academic. This thesis would not have been the same without his countless useful remarks, especially towards the end of my candidature.

I would also like to thank the research team members whom greatly influenced my PhD project. Dr. Serguei Rosskomakhine spent countless number of hours with me sharing his vast knowledge and skills in algorithm implementation. I also
thoroughly enjoyed research discussions with Dr. Mehdi Foumani whose advice played an instrumental role in my research direction. My gratitude also goes to the other members of the Flinders Hamiltonian Cycle Project, especially Dr. Asghar Moeini, for interesting discussions and useful feedback.

I would not be where I am today without the selfless, limitless and unconditional support of my parents. For as long as I can remember, they have done everything in their power to support my education and development. Their dedication in supporting my education in Australia has been the most valuable gift that I have ever received. My gratitude towards them can not be expressed in words.

Lastly, I wish to thank one of my dearest friends, Mr. Andrew Childs. Throughout the challenges during the period of my candidature, Andrew was always the first source of support. His words of encouragement, relentless brotherly kindness and his desire for mutual growth have made a permanent mark on me. Following a tragic incident, Andrew passed away on November 16th, 2017. I am forever grateful for everything that I learnt from him.
Chapter 1

Introduction

1.1 The Traveling Salesman Problem

Suppose a salesman plans to visit a set of cities. The classical formulation of the Traveling Salesman Problem (TSP) is as follows.

**Definition 1.1.1. Traveling Salesman Problem**

Given a set of cities and a set of distances between the cities, find the route of the minimal length that visits each city exactly once and returns to the initial city.

The TSP is one of the most well-studied problems in combinatorial optimization thanks both to its theoretical importance and its applications in many important fields such as logistics [13], genome sequencing [15, 31] and data clustering [84, 93].

TSP belongs to a computationally difficult class of problems called \( \mathcal{NP} \)-hard [99]. We say that an algorithm has *time complexity* \( O(f(n)) \) if, for an input of size \( n \), the algorithm is guaranteed to terminate within \( O(f(n)) \) time steps of some fixed duration. Accordingly, the best known guaranteed running time for a solution to
TSP is $O(n^22^n)$, where $n$ is the number of cities. This bound was obtained by an algorithm developed by Held and Karp [60] in 1962, and it has not been improved since. Unfortunately, it is not computationally practical to solve TSP of sizes beyond tens of cities using the algorithms that guarantee the $O(n^22^n)$ bound. Nevertheless, despite the lack of progress on worst-case running time bounds, significant progress has been made in practical approaches to solving instances of TSP. Indeed, modern algorithms have found the optimal solution for problems with tens of thousands of cities [8]. In comparison, in the early 1970s, it was only possible to solve problems in the order of tens of cities. The TSPLIB library [104] includes many of the benchmark problems studied since the 1950s. For more on some of the notable advances in solving the TSP we refer the interested reader to [33, 50, 51, 39].

While verifying the optimality of a tour for graphs containing more than tens of thousands of nodes is currently impractical, heuristic solutions have succeeded at finding near-optimal solutions for problems with millions of cities. For instance, the best solution discovered by the LKH algorithm [62] for the World TSP challenge [125], which includes around 2 million cities around the earth, is known to be at most 0.0474% longer than an optimal tour. A well-known benchmark set “DIMACS TSP challenge” contains problems of size up to 10 million nodes, and all of the problems have known tours which have been proved to be within 1% of an optimal tour [71]. A genetic algorithm developed by Nagata [96] holds the record for some of the benchmark examples on TSP Art Instances [72] containing up to 200,000 nodes.

1.2 Mathematical Preliminaries

The TSP may be considered in the context of graph theory. Accordingly, this section includes a brief introduction of some of the useful graph theoretic concepts that are relevant to the thesis. For a comprehensive introduction to graph theory,
we refer the reader to [55].

**Definition 1.2.1. Undirected graph**
A graph $G = (V, E)$ is defined as a finite set of nodes $V$, and a prescribed set $E$ consisting of unordered pairs of distinct nodes of $V$.

The elements of $E$ in an undirected graph $G = (V, E)$ are called edges. An edge $(u, v)$ represents a bidirectional link between its endnodes, $u$ and $v$.

**Definition 1.2.2. Directed graph**
A directed graph or digraph $G = (V, E)$ is defined as a finite set of nodes $V$, and a prescribed set $E$ consisting of ordered pairs of distinct nodes of $V$.

The edges (elements of $E$) in a directed graph $G = (V, E)$ are said to be directed. A directed edge $(u, v)$ represents a link from node $u$ to node $v$. Equivalently, a directed edge $u, v$ starts at $u$ and ends at $v$.

An object that is either an undirected graph, or a directed graph may be referred to as a graph. More general definitions of graphs exist; for example by allowing

![Figure 1.1: An example of an undirected 10 node graph. The Hamiltonian cycle is outlined by green dashed lines.](image)
multiple edges between a pair of nodes or allowing an edge with identical endpoints (i.e., a loop). However, in this thesis, we only use graph objects as defined in Definitions 1.2.1 and 1.2.2.

**Definition 1.2.3. Size of a graph**

The size of a graph $G = (V, E)$ refers to the number of nodes $|V|$.

The weight of an edge $(u, v)$ is a constant value associated with the edge and it is denoted by $\omega(u, v)$. A definition of a graph for which the edges are weighted is as follows.

**Definition 1.2.4. Weighted graph**

Consider a graph $G = (V, E)$ where $V$ and $E$ are the sets of nodes and edges of $G$ respectively. Suppose a function $\omega : E \to \mathbb{R}$ is defined that maps every edge $(u, v) \in E$ to its corresponding weight $\omega(u, v)$. Then $G$ is a weighted graph with the weight function $\omega$. The weighted graph may also be denoted by $G = (V, E, \omega)$.

**Definition 1.2.5. Symmetric and asymmetric weights**

Graph $G$ is said to have symmetric weights if for every $(u, v) \in E$, then $(v, u) \in E$ and $\omega(u, v) = \omega(v, u)$. If a graph does not have symmetric weights, we say that it has asymmetric weights.

In this thesis, asymmetric graphs are always represented by directed graphs.

**Definition 1.2.6. Complete graph**

A graph $G = (V, E)$ is a complete graph if $(u, v) \in E$ for every pair of distinct nodes $u, v \in V$.

In a complete graph, each node is linked to all other $n - 1$ nodes. It is obvious
that a complete graph contains the maximal number of edges $n^2 - n$.

**Definition 1.2.7. Degree**

Degree of a node $u$ in a graph $G$ is defined to be the number of edges where $u$ is an endpoint.

If the number of edges $|E|$ is close to the maximal number of edges, the graph is called *dense*. On the other hand, if there the number of edges is small relative to the maximal number of edges, the graph is called *sparse*. While the terms dense graphs and sparse graphs are commonly used in the literature, often the usage is not mathematically rigorous. For a single graph, it is difficult to define an objective measure of whether the number of edges are large or small. A formal definition should be considered in the context of a family of arbitrarily large graphs. For an infinite family of graphs of arbitrarily large sizes, we consider the family of graphs to be a *sparse family of graphs* if

$$\lim_{n \to \infty} \frac{\max(|E|)}{n^2} = 0,$$

where $\max(|E|)$ is the maximal number of edges for a graph of size $n$ in the family of graph. Using this definition, for sufficiently large graphs, the ratio of edges to the maximal number of edges is small. For example, for constant $d$, the family of graphs with maximal degree $d$ is a sparse family of graphs.

**Definition 1.2.8. Cluster**

A *cluster* of nodes $C$ of a graph $G$ is a subset of $V$.

**Definition 1.2.9. Subgraph**

A graph $S = (V_S, E_S)$ is said to be a *subgraph* of a graph $G = (V, E)$ if $V_S \subset V$ and $E_S \subset E$. 

5
**Definition 1.2.10. Vertex-induced subgraph**

Consider a cluster of nodes $C$ of a graph $G$. The *vertex-induced subgraph* of $G$ induced by $C$ is the graph $S_C = (C,E_C)$, where $E_C = \{(u,v) \mid (u,v) \in E, u,v \in C\}$. In other words, the vertex-induced subgraph of $C$ is the subgraph of $G$ that contains all nodes of $C$, and all edges of $G$ which link only the nodes of $C$.

**Definition 1.2.11. Path**

A path of length $k$ in a graph $G = (V,E)$ is a sequence of distinct nodes $(v_1, v_2, \ldots, v_k)$, where $(v_i, v_{i+1}) \in E$ for $i = 1, \ldots, k - 1$.

The path $(v_1, v_2, \ldots, v_k)$ is said to *start from* $v_1$ and *end in* $v_k$.

**Definition 1.2.12. Cycle**

A cycle of length $k \geq 3$ is a sequence of nodes $(v_1, v_2, \ldots, v_k, v_1)$ such that $v_1, v_2, \ldots, v_k$ are distinct nodes. For every cycle $(v_1, v_2, \ldots, v_k, v_1)$, there is a corresponding set of edges $\{(v_i, v_{i+1}) \mid i = 1, \ldots, k\} \cup \{(v_k, v_1)\}$. A *cycle of graph* $G = (V,E)$ is a cycle such that its corresponding set of edges is a subset of $E$.

**Definition 1.2.13. Hamiltonian cycle**

A *Hamiltonian cycle* of a graph $G = (V,E)$ is a cycle of $G$ with length $|V|$. That is, a Hamiltonian cycle traverses all vertices of $V$.

**Definition 1.2.14. Length of a Hamiltonian cycle**

Consider a weighted graph $G = (V,E,\omega)$. Suppose $T$ is the set of edges that correspond to a Hamiltonian cycle. The *length of the Hamiltonian cycle* is defined to be

$$\text{len}(T) = \sum_{(u,v) \in T} \omega(u,v).$$
Definition 1.2.15. *Penalty edge*

If for \( u, v \in V \), an edge \((u,v) \not\in E\), the edge is referred as a *penalty edge*.

Definition 1.2.16. *Tour*

A *tour* \( T \) of a graph \( G = (V,E) \) is a set of edges of \( G \) that form a cycle of size \( n \).

By Definition 1.2.12 of a cycle, there exists a sequence of nodes that corresponds to any tour. This provides an alternative sequence representation of a tour. For example, a tour represented by the set \{ \( (v_1, v_2), (v_2, v_3), \ldots (v_{n-1}, v_n), (v_n, v_1) \) \} may also be represented as a sequence of nodes \( (v_1, v_2, v_3, \ldots, v_k) \).

Definition 1.2.17. *Gap*

Consider a tour \( T \) of a graph \( G = (V,E) \). If \((u,v) \in T \) but \((u,v) \not\in E\) (i.e., \((u,v)\) is a penalty edge), then \((u,v)\) is called a *gap* on \( T \).

Note that in the notation used in this thesis, there is a difference between a Hamiltonian cycle and a tour. In contrast to a Hamiltonian cycle, a tour may contain penalty edges. Accordingly, a Hamiltonian cycle may also be defined as a tour that does not contain any gaps.

We conclude this section by embedding the TSP into a weighted graph. Construct a graph \( G = (V,E,\omega) \) by representing every city by a node, and representing the travel between each pair of cities by an edge. For example, the travel from city \( u \) to city \( v \) is represented by the edge \((u,v)\), and the distance of travel between the pair of cities is denoted by \( \omega(u,v) \). Then the TSP may be formally defined as follows.

Definition 1.2.18. TSP on a weighted graph \( G \)

Given a weighted graph \( G = (V,E,\omega) \), find a Hamiltonian cycle of minimal length.
1.3 Solving the Traveling Salesman Problems

A search that guarantees an optimal tour for a TSP has a computational bound of at least $O(n^{2^n})$, and is therefore impractical for any problem larger than only tens of nodes. However, there is an alternative to this method, which is not guaranteed to work in a reasonable amount of time, but is it often far more effective than the guaranteed method. The alternative approach has two main components; a process that finds an upper bound, and a process that finds a corresponding lower bound. The upper bound value, which is obtained by calculating the length of a known tour, is then compared to the lower bound. If the upper bound and lower bound values meet, then the tour is verified to be optimal. For example, suppose a heuristic approach finds a tour with the value $L$. Also, suppose that $L$ is also shown to be a lower bound for an optimal tour using an efficient algorithm. In this case, the tour is confirmed to be optimal without the need for an exhaustive search. It should be noted that, sometimes the process of finding a lower bound may itself find a valid tour; in such a case, the tour must be optimal. However, generally speaking, combining the two independent pieces is the most effective strategy.

1.3.1 Lower Bounds for TSP

To find an exact solution for TSP, researchers often use an integer linear programming (ILP) formulation of TSP. Accordingly, the most common method for finding a lower bound is to solve a relaxation of the ILP formulation. There are many challenges to overcome when finding a good lower bound. To briefly overview these challenges, let us consider one of the most well-known ILP formulations of
TSP, namely the TSP formulation with sub-tour elimination constraints. Suppose $\omega(i, j)$ is the length of edge $(i, j)$. Let

$$ x_{ij} = \begin{cases} 
1 & \text{if edge } (i, j) \text{ is on the tour}, \\
0 & \text{otherwise}.
\end{cases} $$

Then the objective function is

$$ \min \sum_{i \in V} \sum_{j \in V \setminus \{i\}} x_{ij} \omega(i, j) : \text{minimize the length of the tour} $$

Subject to

$$ \begin{cases} 
C1: \sum_{j \in V \setminus \{i\}} x_{ij} = 2, & \forall i \in V, \\
C2: \sum_{i,j \in S, j \neq i} x_{ij} \leq |S| - 1, & \forall S \subset V, S \neq \emptyset,
\end{cases} $$

where $V$ is the set of all nodes (i.e., cities). The number of constraints of type $C1$ and $C2$ is equal to the number of nodes, and the number of proper subsets of nodes respectively. Therefore there are $n$ constraints of type $C1$ and $2^n - 2$ constraints of type $C2$. Constraints $C2$ are the so-called subtour elimination constraints.

The first challenge to solving TSP via this formulation is that the computational complexity of solving the ILP formulation is still $\mathcal{NP}$-hard. To avoid this difficulty, LP relaxations of ILP are solved first, and their solutions provide a lower bound for the length of an optimal tour. Second, the number of sub-tour elimination constraints grows exponentially relative to the size of the problem. Therefore, it is practically impossible to consider all of the constraints. Instead, it is desirable to use strategies that include only a small subset constraints, while still obtaining reasonable lower bounds. The third challenge is that considering all edges of the underlying graph also results in slow computations by creating dense LPs.

The search space may be restricted further by only considering a subset of edges, which is equivalent to forcing a value $x_{ij} = 0$ for all edges that are not in the
subset. However, since we do not know the right subset of edges to choose from, the subset of edges should be a dynamic set. In other words, it is desirable to be able to eliminate existing edges or add other edges to the dynamic set with the aim of improving the lower bound for TSP.

Currently, the best algorithm that effectively implements the above strategy to solve TSP is Concorde [7]. Concorde also uses a branch and bound implementation to eventually achieve an ILP solution (guaranteed in exponential time). Remarkably, Concorde has been able to successfully verify optimal solutions for problems containing tens of thousands of nodes.

1.3.2 Upper Bounds for TSP: Tour Improvement Heuristics

The length of any tour serves as a upper bound for the length of an optimal tour. If one reviews the best performance records on TSP benchmark sets over the past thirty years, the success of tour improvement algorithms is highly evident. Specifically, variations of the seminal Lin-Kernighan algorithm [88] have been at the forefront of the best heuristic solutions to large TSP instances. We discuss two important components of these algorithms, namely, sparsification and tour improvement techniques.

Sparsification

Since an an optimal tour contains $n$ edges, and a complete graph contains $n^2 - n$ edges, it stands to reason that many of the edges can be removed without changing the solution of the TSP problem. The process of removing edges from the graph with the goal of reducing the search space is called sparsification. An effective sparsification can significantly reduce the number of edges, possibly from
$O(n^2)$ to $O(n)$. The challenge of sparsification is that sometimes the sparsification process may remove edges of an optimal tour from the graph. Therefore the quality of sparsification depends not only on how many edges it removes, but also on preserving the edges of an optimal tour. The method of sparsification in the original Lin-Kernighan algorithm is a greedy process, retaining only the five shortest edges incident to a city. The greedy method for sparsification becomes substantially less effective as the size of the graph increases. The reason is that the edges of an optimal tour may have many edges with larger costs than the five shortest edges incident to a city[98].

Heuristic sparsification methods have improved significantly in recent years, and have had a remarkable impact on the quality of upper bound solutions. For example, Helsgaun [62] introduced a concept called $\alpha$-nearness which ranked the edges after solving a spanning 1-tree problem. This alternative sparsification method resulted in significant improvement in the algorithmic performance of the Lin-Kernighan algorithm. The downside of such a sparsification method is that it may still remove edges of an optimal tour from the graph. On the other hand, some methods are guaranteed to only remove edges that are not on an optimal tour [68]. Even though it is experimentally shown that they perform well on certain classes of TSP instances, for example on problems with Euclidean distances, it is not guaranteed that the number of edges for a given TSP problem is significantly reduced.

**Tour Improvement Techniques**

After limiting the search space to a sparse graph, tour improvement techniques may be used to transform an initial tour to a better tour. In the 1960s, heuristic tour improvement techniques were introduced to solve problems where exact methods were not practical. Lin [87] introduced transformations called $\lambda$-opt that exchanged $\lambda$ edges on the current tour with $\lambda$ other edges such that the resultant
tour had a better (lower) value. Since the number of possible $\lambda$-opt transformation grows exponentially as $\lambda$ grows, $\lambda$ needs to be bounded for the algorithm to have a practical run-time. Usually $\lambda$ is bounded by 2 or 3, but some studies have experimented with values 4 and 5 [27]. As the size of the problem grows, the necessity of exchanges with higher values of $\lambda$ becomes more apparent.

To overcome this issue, Lin and Kernighan [88] introduced an algorithm that allowed a substantial improvement in heuristic solutions to TSP. The central idea of the Lin-Kernighan algorithm was the introduction of transformations that allowed efficient consideration of certain $\lambda$-opt operations where $\lambda$ is not bounded. These transformations are commonly referred to as sequential transformations. Of course, the drawback of only relying on sequential transformations is that there are situations where non-sequential exchanges are required to improve a tour. Implementation of the Lin-Kernighan algorithm requires design choices that have a significant impact on the performance of the algorithm. Since the introduction of the basic algorithm in 1971, variations of the algorithm have been developed to tackle ever larger problems [91, 9]. These algorithms have often been at the forefront of the best performance for problems too large for exact algorithms. Currently, the best implementation of the Lin-Kernighan algorithm is widely regarded to be Helsgaun’s LKH [62], which hold various records in the most well-known TSP benchmark sets.

1.4 Variations of the Traveling Salesman Problem

Variations and extensions of the TSP have also been a vibrant topic of study in the literature [53]. For example, suppose the set of cities is partitioned into a number of clusters, and the traveling salesman must visit all of the nodes within each cluster consecutively. This problem is called the Clustered Traveling Salesman
Problem (CTSP) and various solution methods have been developed to address this variation of TSP [5, 12, 38, 94]. Another example is the Generalized Traveling Salesman Problem (GTSP) where the number of cities is also divided into clusters, but the salesman must visit exactly one node in each cluster. The GTSP has also been extensively studied [81, 73, 74, 113].

There are two primary reasons for considering variations of TSP. The first motivation is their theoretical and algorithmic significance. For example, The Hamiltonian Cycle Problem (HCP) is a special case of TSP and it has theoretical significance in the context of complexity theory. The HCP belongs to a class of problems called \( \mathcal{NP} \)-complete. Proving the existence or non-existence of an efficient (polynomial-time) algorithm for HCP is equivalent to the famous \( \mathcal{P} \) versus \( \mathcal{NP} \) problem, widely regarded to be one of the most important problems in mathematics and computer science [29]. In addition, being a difficult special case of the TSP, algorithmic advances for solving the HCP potentially have implications for solving the TSP as well.

Second, considering the problems that are generalizations of TSP have the potential for array of applications. For example, CTSP and GTSP have been applied to postal delivery systems [82], warehouse logistics [97] and integrated circuit testing [82]. It stand to reason that, if a variation of TSP includes the classical TSP as a special case, there is a higher degree of flexibility for modeling.

The chapters of this thesis are designated to three particular variations of the Traveling Salesman Problem. These problems are studied in the context of algorithmic solutions and potential applications.

**Definition 1.4.1.** *Hamiltonian Cycle Problem (HCP)*

Given a graph \( G = (V, E) \), find a cycle of size \( |V| \) (i.e., a Hamiltonian cycle).
The HCP is equivalent to a special case of the TSP where the weights are binary values. Since it encompasses significant computational difficulties, the study of the HCP is potentially enlightening for understanding the TSP. Its special structure also provides an opportunity to create benchmarking instances for TSP and to study algorithmic solutions for special structures in TSP.

**Definition 1.4.2.** *Sparse Traveling Salesman Problem (STSP)*

Given a graph $G = (V, E, \omega)$, suppose $g_{\min}$ is the minimum number of gaps for any tour of $G$. The STSP is the problem of finding the shortest tour with $g_{\min}$ gaps.

The Sparse Traveling Salesman Problem is a slight variation of TSP where the aim is to find the shortest tour with minimal gaps. Algorithmic approaches to TSP are not only applicable to TSP on sparse graphs, but also to graphs that may not contain any Hamiltonian cycles. For example, the TSP formulation of the DNA-assembly problem often produces a graph with no Hamiltonian cycles. Nevertheless, a desirable solution would have the minimal number of gaps.

**Definition 1.4.3.** *Clustered Generalized Traveling Salesman Problem (CGTSP)*

Consider a weighted graph $G = (V, E, \omega)$. Suppose the nodes are partitioned into mutually exclusive clusters, and each cluster is further partitioned into mutually exclusive sets of nodes called subclusters. Then the CGTSP is the problem of finding the shortest cycle that satisfied the following:

1. the cycle visits exactly one node per subcluster, and,
2. within each cluster, all of the visited nodes are visited consecutively.

The CGTSP is an extension of the TSP. It is a combination (and a generalization) of two variations of TSP, namely CTSP and GTSP. The CGTSP offers a higher
degree of flexibility for modeling compared to a classical TSP model, and is a promising tool for applications in logistics.
Chapter 2

A Novel Tour Improvement Approach for The Hamiltonian Cycle Problem

2.1 Introduction

Recall from Definition 1.4.1 that a Hamiltonian cycle (HC) in a graph is a cycle of a graph that includes all nodes of the graph. The Hamiltonian Cycle Problem (HCP) is the problem of finding one HC in a graph, or determining that none exists. The HCP is equivalent to the special case of TSP where all of the weights are binary values. It is also closely related to the TSP on sparse graphs, as the optimal tour of any instance of TSP must be a Hamiltonian cycle. Since the HCP is an $\mathcal{NP}$-complete problem, it encapsulates some of the underlying difficulties of solving TSP on sparse graphs.

The work contained in this chapter is the continuation of the work in Baniasadi et al. [10], which introduced a heuristic algorithm, called the Snake and Ladders Heuristic (SLH), for solving the HCP. The performance of SLH on HCP is on a par with the state-of-the-art TSP and HCP algorithms like Concorde and LKH, and
there are many cases where SLH outperforms these algorithms. The performance of SLH is particularly impressive on difficult HCP instances where the number of Hamiltonian cycles is extremely small.

In this chapter, I describe a HCP heuristic, based largely on new refinements for SLH. However, three different aspects differentiate this work to that of [10]. Many of the algorithmic concepts of [10] were discovered by considering many approaches and selecting the final methods through trial and error. In contrast, here I abstract out the key theoretical concepts in order to understand how and why SLH is such an effective algorithm. These concepts may be useful for fine-tuning the algorithm implementation, finding further improvements to the method, and for generalizing the algorithmic methods to other problems. Second, the algorithm presented in this chapter is designed to work as an exact algorithm rather than a heuristic method. Finally, SLH as prescribed in [10] relies on multiple stages that apply different heuristic strategies. Here, the solution strategy is unified into a single strategy consisting only of the most important algorithmic principles.

The remainder of this chapter is organised as follows. For the remainder of this section, we introduce the required notation, and give a brief summary as to how SLH works. Then, in Section 2.2, we introduce the concept of a blind improvement space, which is a prescription for a set of transformations which is guaranteed to contain at least one which improves the tour by a particular measure. In Section 2.3, we identify three main principles which should be adhered to by an algorithm which utilises transformations from a blind improvement space, and we then discuss each of these in detail in Sections 2.4–2.6 respectively. Finally, in Section 2.7, we present the algorithm which combines each of these concepts into a solver for HCP. The performance of the algorithm is subsequently benchmarked in Chapter 3.
2.1.1 Notation

Consider an instance of HCP defined on an undirected graph $G = (V, E)$, where $V$ is a set of $n$ nodes, and $E$ is a set of edges representing the links between the nodes. If an edge $(v, w) \notin E$, it is referred as a penalty edge. A tour $T$ is a set of edges that form a simple cycle of size $n$. We note that in most of the notable work on tour improvement algorithms since the 1960s [87, 88], the concept of a tour has been (implicitly or explicitly) defined on a complete graph. In order to be consistent with the literature on tour improvement algorithms, we permit a tour to contain penalty edges, and we call a penalty edges on a tour a gap. As a result, even though a tour always corresponds to a Hamiltonian cycle in a complete graph, this is not necessarily the case for a sparse graph. In this context, the HCP is the problem of finding a tour that does not contain any gaps.

Consider a set of edges $X$ such that $X \subset T$, and another set of edges $Y$ such that $Y \cap X = \emptyset$ and $\text{card}(Y) = \text{card}(X) = k$. Also let $T' = (T \setminus X) \cup Y$. If $T'$ is a valid tour (i.e., it contains no sub-tours), then the transformation of $T$ to $T'$ is known as a feasible $k$-exchange.

A segment of a tour is a path that only contains the edges on the tour. Consider three distinct nodes $a, b, c \in V$. Since $T$ is a simple cycle, there are exactly two segments of the tour going from $a$ to $b$ on the tour. One of these two paths must contain $c$, and the other one does not contain $c$. Then a segment of a tour may be unambiguously represented as triplet $(a, c, b)$. The triplet representation of a segment is an equivalence class of triplets where $c$ is any of the intermediary nodes on the path between $a$ and $b$. We also define a complement of a segment $(a, c, b)^c$ to be the path between $a$ and $b$ on the tour that does not contain $c$. 
2.1.2 Highlights of the Snakes and Ladders Heuristics

Before expanding the concepts in Baniasadi et al. [10], I first present a short summary of the SLH algorithm. Similar to other tour improvement algorithms, SLH performs a series of transformations with the aim of obtaining a Hamiltonian cycle. At each iteration, an available transformation is chosen from a list of prescribed transformations, which maps the current tour to the next tour.

Transformations of SLH can be described as compositions of two generator transformations, specifically the following two transformations:

**Definition 2.1.1.** $\gamma[y,x,a](T)$ transformation.

Consider a tour $T = \{(x, y), (a, b), \ldots\}$ where $b$ is on the $(x, y, a)$ segment. The transformation $\gamma[y, x, a](T)$ maps tour $T$ to tour $T' = (T \setminus \{(x, y), (a, b)\}) \cup \{(x, a), (b, y)\}$.

Note that in the above definition of $\gamma$, the choice of $b$ is forced by the selection of $[y, x, a]$ because there exists a unique node $b$ on the $(x, y, a)$ segment where $(a, b) \in T$. The transformation $\gamma[y, x, a]$ may be visualized as flipping the segment $(y, x, a)^c$, when representing a tour as a circular ordering of the nodes, as illustrated in Figure 2.1.

![Figure 2.1: $\gamma[y, x, a](T)$ transformation.](image-url)
**Definition 2.1.2.** \( \kappa[y,x,a,b,c](T) \) transformation

Consider a tour \( T = \{ (y,x),(a,e),(d,b),(c,f), \ldots \} \) where \( b \) is on the \( (y,x,a) \) segment, \( e \) and \( c \) are on the \( (y,x,a)c \) segment, \( d \) is on the \( (y,x,b) \) segment, and \( f \) is on the \( (a,e,c) \) segment. Then for \( X = \{ (y,x),(a,e),(d,b),(c,f) \} \) and \( Y = \{ (a,x),(e,d),(b,c),(f,y) \} \), transformation \( \kappa[y,x,a,b,c](T) \) maps \( T \) to \( T' = (T \setminus X) \cup Y \).

In the definition of \( \kappa[y,x,a,b,c](T) \) transformation, the choices for \( e, d \) and \( f \) are forced. A visual representation of \( \kappa[y,x,a,b,c](T) \) is shown in Figure 2.2.

![Figure 2.2: \( \kappa \) transformation \( \kappa[y,x,a,b,c](T) \).](image)

In the context of HCP, the value of the tour improves when the number of gaps is decreased. If there are no gaps on the tour, then the tour is a Hamiltonian cycle. Therefore, the number of gaps \( g(T) \) is a natural measure of the value of the tour \( T \). Accordingly, we define three types of transformations from a tour \( T \) to any another tour \( T' \):

- A transformation is called a **closing** transformation if \( g(T') < g(T) \).
- A transformation is a **floating** transformation if \( g(T') \leq g(T) \).
- A transformation is called an **opening** transformation when the transformation may (but not necessarily) increase the number of gaps.
Note that these definitions are *not* mutually exclusive. The transformations that we will discuss in this chapter are defined parametrically, and the above terms are defined to distinguish whether the key conditions are met.

It was proven in [10] that, for any given tour $T$, there exists a sequence of transformations involving only $\gamma$ and $\varkappa$ that can transform $T$ to a Hamiltonian cycle (if one exists). Therefore, all of the transformations used in SLH are prescribed compositions of $\gamma$ and $\varkappa$ transformations involving up to 5 exchanges. Accordingly, there exist multiple closing and floating transformations, that may be performed on a tour with a gap when certain conditions are met. There is only one opening transformation, of the form $\varkappa[y, x, a, b, c](T)$, where $(y, x)$ is a gap and $(x, a), (b, c) \in E$.

In essence, SLH performs a sequence of $\gamma$ and $\varkappa$ transformations with the intention of finding a Hamiltonian cycle.

SLH utilises four strategies for reducing the number of gaps. These strategies are implemented in multiple stages; if no Hamiltonian cycle is found in one stage, SLH moves on to the next stage. If all stages fail to find a Hamiltonian cycle, SLH outputs the best visited tour, with the minimal number of gaps, and declares the graph to be “likely non-Hamiltonian”. A brief review of the function of each stage of SLH is as follows:

**Stage 0.** Only closing transformations are permitted on the current tour.

**Stage 1.** Floating transformations are permitted on the current tour. The space of transformations in this stage is limited by the following:

- Any floating transformation exchanges at least one gap on the current tour.

  When a transformation is performed, this gap is saved in a list of gaps. If a gap is in the list of gaps, a transformation that exchanges the gap is not permitted. The list of gaps is cleared every-time an improvement on the best tour is found. This rule limits the number of consecutive transformations.
without an improvement to $n^2$ during stage 1.

- If there are no gaps (i.e., the tour is a Hamiltonian cycle), return the best tour.

**Stage 2.** Save the first tour at stage 2. Perform one opening transformation on the first tour which has not been attempted before, and repeat stage 1. If an improvement on the best tour is found, return to stage 1. If no improvement on the best tour is found, then repeat this process on the first tour with a different opening transformation. If there are no more opening transformations available on the first tour of stage 2, go to stage 3.

**Stage 3.** First attempt to perform closing transformations on the current tour. If no closing transformation is available on the current tour, perform an opening transformation. The space of transformations in this stage is limited by the following:

- Every time a transformation is performed, save the current tour to a list of visited tours. A transformation that maps the tour to a list of visited tours is not permitted.

- If an improvement to the best tour is found, clear the list of visited tours, and return to stage 1.

- If the list of visited tours contains $n^3$ tours, return the best tour.

For each tour at each stage, there may be multiple available transformations. To minimize the time spent on finding a transformation, the first applicable transformation found is performed.
2.2 Tour Improvement and the Blind Improvement Space

SLH is closely related to a class of algorithms called tour improvement algorithms. Tour improvement algorithms start from an initial tour, which could be constructed from any permutation of nodes, and they try to improve the tour by transforming the tour to a better tour. The criteria for improvement in almost all tour improvement algorithms is defined as moving from a tour to another tour with a lower value. A feasible $k$–exchange that transforms a tour $T$ to a shorter tour $T'$ is called a $k$–opt transformation. A tour improvement algorithm searches the space of $k$–exchanges, or some subset of the $k$–exchanges space, to find a $k$–opt transformation.

It is possible that a sub-optimal tour $T$ does not share any edges with any improved tour $T'$. In these situations, an $n$–exchange transformation on $T$ is the only way to improve the tour. For a naive tour improvement algorithm, this is a serious computational difficulty because there are $O(n!)$ transformations to choose from.

There are a few ways to limit the size of the search space for a transformation. The first attempts in designing tour improvement algorithms placed bounds on $k$ when searching for a $k$-exchange. For example, early implementations only considered 2–opt and 3–opt transformations [87]. Later studies also considered 4–opt and 5–opt transformations [27]. However, since the space of transformations grows exponentially with respect to $k$, this method is very limited in its scope for further extensions. The major breakthrough in tour improvement algorithms was the introduction of the Lin-Kernighan heuristic [88] in the 1970s, which enabled a fast search for $k$–exchanges for variable $k$, by limiting consideration to a special type of $k$–exchanges called sequential exchanges. To this day, the Lin-Kernighan type heuristics are at the forefront of upper-bound solutions to TSP and its variations [9, 62, 64].
Nevertheless, by excluding the space of transformations that are not sequential, Lin-Kernighan type heuristics have an obvious weakness for the instances where sequential exchanges are insufficient. Furthermore, it is possible to reach a tour with a value that is very close to the optimal tour, while having very few intersections with the edges of the optimal tour. In such a case, the algorithm may get stuck at a local minimum where it is very difficult to proceed to the global minimum. For this reason, the value of the tour is not a perfect measure of improvement if our objective is to obtain an optimal tour.

In [10], an alternative perspective on the concept of tour improvement was proposed.

**Definition 2.2.1. Distance of between a tour and an optimal tour**

Denote by $T^{\text{opt}}$, an optimal tour of the graph. For any tour $T$ we define the distance between $T$ and $T^{\text{opt}}$ as

$$\text{dist}(T, T^{\text{opt}}) = |T \setminus T^{\text{opt}}|.$$

**Definition 2.2.2. Distance improving transformation**

Consider a transformation of $T$ to another tour $T'$ such that $\text{dist}(T', T^{\text{opt}}) < \text{dist}(T, T^{\text{opt}})$. Then we call the transformation of $T$ to $T'$ a distance improving transformation.

In this chapter, I further expand on this alternative notion of improvement. Of course, since $T^{\text{opt}}$ is not known to a TSP solver, the solver cannot determine whether a particular transformation is a distance improving transformation or not. However, for any tour of a given graph, a set of transformations can be defined such that at least one transformation in the set is guaranteed to be a distance improving transformation. We will show that it is possible to define
prescribed ways to create such a set of transformations for any non-optimal tour of any graph.

Definition 2.2.3. **Blind improvement space (BIS)**

Suppose $F$ is a map that associates any graph $G$ and any non-optimal tour $T$ of $G$, to a finite set of transformations $B$, where at least one transformation in $B$ is a distance improving transformation. $B = F(T,G)$ is called the **blind improvement space (BIS)** of $T$ in $G$ and $F$ is called a **BIS prescription** or a **BIS map**.

Instead of the term BIS of $T$ in $G$, we may use the shortened term BIS of $T$, where there is no ambiguity about the graph $G$. Any set of transformations that contain at least one distance improving transformation is not necessarily a BIS. A BIS must be created by a BIS map which must satisfy a strict condition. That is, a BIS map must be a general prescription for creating sets of transformations, which **irrespective of the variations in graph structures**, map any non-optimal tour of a graph to a finite set of transformations that contain at least one distance improving. For example, it is trivial that the set of all possible transformations on $T$ of $G$ constitutes a BIS for any non-optimal tour $T$. Of course, this trivial example is too large to be useful for algorithmic purposes. However, we will show that it is possible to define BIS prescriptions that are appropriate for algorithmic purposes.

Most tour improvement algorithms find and perform $k$–opt transformations, which are $k$–exchanges that improve the value of the tour. In contrast, an individual transformation of a BIS based algorithm may or may not immediately lead to a lower value tour. Rather, the central idea in a BIS based algorithm is clearly define, in advance, a BIS prescription in such a way the search space can be efficiently searched, and then choose one transformation in a BIS in each iteration. In this
way, the algorithmic approach of SLH is fundamentally different from algorithms relying on $k$-opt transformations.

### 2.3 Design Fundamentals of a BIS based Algorithm

There are two important measures of improvement in our algorithmic design; the value of the tour and distance to the optimal tour. A combination of both measures would potentially provide the advantages of both measures of improvements. The value of the tour is a good measure for obtaining a local minimum, and selecting transformations from a BIS allows the algorithm to avoid getting stuck in a local minimum. Together, they potentially increase the chances of obtaining an optimal tour.

I now discuss the algorithmic approach in the context of the HCP, or equivalently, finding a Hamiltonian cycle in a sparse graph. These concepts have been developed with the intention of generalizing them for the TSP. Demonstrating their value for solving instances of HCP will serve as a proof of the merit for further investigation in the context of TSP.

The foundation of the algorithmic approach is the following three key general principles.

**Principle 1:** The use of a blind improvement space: For any current tour, the algorithm obtains the next tour by performing one transformation which is chosen from a BIS of the tour in the graph.

**Principle 2:** Transformations that increase the number of gaps are used only as a last resort: a transformation which increases the number of gaps is permitted only if all other options in the BIS of the tour are unavailable.
**Principle 3:** Prioritizing transformations with lower computational cost: Among floating and closing transformations, the search for a transformation to obtain the next tour is designed to minimize computations for obtaining the next tour.

Principle 1 limits the space of transformations that the algorithm considers. Principles 2 and 3 are the guidelines for prioritizing some transformations over others, within the the space that the algorithm considers.

The remaining parts of this chapter are dedicated to developing and refining the concept of BIS using the above principles. In Section 2.4, I discuss the fundamentals of the concept of BIS and introduce a basic BIS prescription for any non-optimal tour \( T \). I then discuss modifications to the basic BIS to help restrict the usage of opening transformations in Section 2.5. In Section 2.6, I explore further refinements of the BIS prescription designed to minimize computations necessary to select transformations. Finally, in Section 2.7, I describe the new algorithm for solving the HCP.

### 2.4 Principle 1: Constructing a Blind Improvement Space

The first challenge is to construct a BIS prescription which is computationally practical to construct. It is trivial that the space of all \( k \)–exchanges where \( k \leq n \) constitutes a BIS for any non-optimal tour. However, this space contains \( O(n!) \) transformations, and hence it is computationally impractical to search for a transformation in the BIS of all \( k \)–exchanges. Accordingly, our first objective is to construct a relatively small BIS, so that it may be searched in a reasonable amount of time.
In the pursuit of constructing a suitable BIS prescription, let us start by considering only the 2-exchange transformations. We will first show that any feasible 2-exchange can be represented by a $\gamma$ transformation. Then we will show that the space of all possible 2-exchanges does not always constitute a BIS for a tour.

**Lemma 2.4.1.** Let $T = \{(x, y), (a, b), (c, a) \ldots\}$ where $b$ is on the $(y, x, a)$ segment. Let,

- $T' = (T \setminus \{(y, x), (a, b)\}) \cup \{(x, a), (b, y)\}$, and
- $T_2' = (T \setminus \{(y, x), (b, a)\}) \cup \{(x, b), (a, y)\}$.

Then, $T'$ is a valid tour, and $T_2'$ is not a valid tour.

**Proof.** Since $b$ is on the $(y, x, a)$ segment, then removing $(x, y)$ and $(a, b)$ from $T$ creates two disconnected paths of the form $(x, \ldots, b)$ and $(y, \ldots, a)$. It is clear that the addition of $(x, b)$ and $(a, y)$ does not connect these two paths in $T_2'$, and hence, $T_2'$ is not a simple cycle of size $n$. The addition of $(x, a)$ and $(b, y)$ however, connects the end of each path to the beginning of the other path, creating a simple cycle of size $n$, i.e., a valid tour. \hfill $\square$

Note that the $\gamma[y, x, a](T)$ transformation maps $T$ to $T'$, which by Lemma 2.4.1 is a valid tour. Therefore, the definition of $\gamma$ is unambiguous and will always output a valid tour. In addition, we can make the following statement with respect to the $\gamma$ transformation.

**Lemma 2.4.2.** Any feasible 2-exchange transformation may be represented as a $\gamma[y, x, a](T)$ transformation.

**Proof.** Suppose the 2-exchange transformation exchanges a set of two edges $X$ of $T$ with a set of two edges $Y$ to create the tour $T'$, i.e, $T' = (T \setminus X) \cup Y$. Let $X = \{(x, y), (a, b)\}$ be any set of two edges on the tour $T$. In $T \setminus X$, the nodes $x, y, a, b$ are each adjacent to exactly one node, while all other nodes are adjacent.
to exactly two nodes. The 2-exchange transformation is feasible, meaning $T'$ is a tour where every node is adjacent to exactly two nodes. Therefore adding the edges of $Y$ to $T \setminus X$ must add edges whose endpoints are $x,y,a$ and $b$. The only two options are $Y = \{(x,a), (b,y)\}$ and $Y = \{(x,b), (a,y)\}$.

Without loss of generality, suppose $b$ is on the $(x,y,a)$ segment. The generality is not lost because if $b$ is not on the $(x,y,a)$ segment, then $a$ must be on the $(x,y,b)$ segment, in which case we may simply change the labeling of $a$ and $b$. By Lemma 2.4.1, $Y = \{(x,a), (b,y)\}$ is a valid option for feasible transformation and the other option is not. Additionally, this transformation is equivalent to $T' = \gamma[y,x,a](T)$. Hence, for any pair of edges on the tour $X = \{(x,y), (a,b)\}$, there exist a unique 2-exchange feasible transformation, and this transformation may be expressed by $\gamma[y,x,a]$ where $b$ is on the $(x,y,a)$ segment. □

Lemma 2.4.3. The set of all feasible 2-exchanges for a non-optimal tour of a graph does not constitute a BIS for the tour.

Proof. We construct a simple counter-example. Let the graph $G = (V,E)$ where $V = \{a,b,c,d,e,f\}$ and $E = \{(a,d), (a,f), (b,c), (b,e), (c,f), (d,e)\}$ and the tour $T = \{(a,b), (b,c), (c,d), (d,e), (e,f), (f,a)\}$. The optimal tour of this graph is $T^{opt} = \{(a,d), (d,e), (e,b), (b,c), (c,f), (f,a)\}$. It is easy to exhaustively check all feasible 2-exchanges to confirm that $\text{dist}(T, T^{opt})$, or equivalently the number of gaps on $T$ for this example, can not be improved by any 2-exchanges. □

It is therefore necessary to look beyond 2-exchange transformations to prescribe a BIS. Next, we consider the two transformations defined below. We will show that, collectively, they form a BIS for a tour. The first transformation is a special type of $\gamma$ transformation.

Definition 2.4.4. Floating $\gamma$ transformation.

Let $T$ be a tour that contains a gap $(y,x) \notin E$. Also suppose let node $a$ be such
that \((x, a) \in E\) but \((x, a) \notin T\). Then the transformation \(\gamma[y, x, a](T)\) is called a floating \(\gamma\) transformation.

The floating \(\gamma\) transformation is displayed in Figure 2.4. Note that we use the term “floating” to designate that the number of gaps may stay the same, or may reduce (if edge \((b, y)\) is present).

**Definition 2.4.5.** Opening \(\psi\) transformation.

Let \(T\) be a tour that contains a gap \((y, x) \notin E\). Consider the nodes \(x, y, a, b, d\) where \((a, d) \in T\) is on the tour, \(d\) is on the \((y, x, a)^c\) segment, and \(b\) is on the \((y, x, a)\) segment. Then the \(\psi[y, x, a, b](T)\) transforms \(T\) to \(T' = (T \setminus X) \cup Y\) for

\[
X = \{(y, x), (a, d), (c, b)\} \quad \text{and} \quad Y = \{(x, a), (d, c), (b, y)\}
\]

where \(c\) is adjacent to \(b\).
on the \((y, x, c)\) segment of \(T\).

The \(\psi\) transformation is a new 3-exchange transformation. It is easy to check that \(\psi[y, x, a, c](T) = \gamma[b, x, a] \circ \gamma[b, c, y] \circ \gamma[a, b, c](T)\). While an opening \(\psi\) transformation is a composition of \(\gamma\) transformations, we note that the individual \(\gamma\) transformations in the composition may not be floating transformations. Hence, the addition of opening \(\psi\) transformations to floating \(\gamma\) transformations in the search space enables transformations to tours which could not have been visited otherwise via only the compositions of floating \(\gamma\) transformations.

![Figure 2.5: Opening \(\psi\) transformation \(\psi[y, x, a, b](T)\).](image)

**Proposition 2.4.6.** Suppose \((x, y) \in T\) but \((x, y) \notin T^{opt}\). Then, either

(i) \(\exists a\) such that \(\gamma[y, x, a](T)\) is a distance improving floating \(\gamma\) transformation, or,

(ii) \(\exists a, b\) such \(\psi[y, x, a, b](T)\) is a distance improving opening \(\psi\) transformation.

**Proof.** In a tour, all nodes are adjacent to exactly two nodes. Then since \((x, y) \in T\) but \((x, y) \notin T^{opt}\), there exist a node \(a\) such that \((x, a) \in T^{opt}\) but \((x, a) \notin T\). There also exists another node \(e \neq x\) such that \((a, e) \in T^{opt}\). There are two possible cases.

**Case 1:** Suppose \((a, e) \in T\) and \(e\) is on the \((y, x, a)\) segment. Note that there exists
another node \(d\) such that \((a, d) \in T\) and \(d\) is on the \((y, x, a)^c\) segment. Tour \(T^{opt}\) can not include all of the edges on the \((x, e, a)\) segment because it also includes \((x, a)\), which would create a subtour. There exists an edge \((b, c) \notin T^{opt}\) where \(b\) and \(c\) are on the segment \((y, x, a)\), and without loss of generality, \(c\) is on the \((y, x, b)\) segment. Consider the transformation \(\psi[y, x, a, b](T)\) which transforms \(T\) to \(T' = (T \setminus X) \cup Y\) for \(X = \{(y, x), (a, d), (c, b)\}\) and \(Y = \{(x, a), (d, c), (b, y)\}\). We know that \((x, y), (a, d), (b, c) \notin T^{opt}\) and \((x, a)\) is in \(T^{opt}\), and all other edges which are present in \(T\) are also present in \(T'\). Therefore, \(\text{dist}(T', T^{opt}) \leq \text{dist}(T', T^{opt}) - 1\), and hence, \(\psi[y, x, a, b](T)\) is a distance improving opening \(\psi\) transformation for this case.

\[ \begin{align*}
\text{Figure 2.6:} & \quad \text{Case 1 in the proof of Proposition 2.4.6. The bold edges are those known to be in } T^{opt}, \text{ and the dashed edges are those known to not be in } T^{opt}. \\
\end{align*} \]

Case 2: Suppose either \((a, e) \notin T\), or that \((a, e) \in T\) but \(e\) does not lie on the \((y, x, a)\) segment (which covers any situation that is not covered by Case 1). It follows that there exists a node \(f\) on the \((y, x, a)\) segment of \(T\) such that \((a, f) \in T\) but \((a, f) \notin T^{opt}\). Then \(\gamma[y, x, a](T)\) transforms \(T\) to \(T' = (T \setminus X) \cup Y\) for \(X = \{(x, y), (a, f)\}\) and \(Y = \{(x, a), (f, y)\}\). We know that \((x, y), (a, f) \notin T^{opt}\) and \((x, a)\) is in \(T^{opt}\), and all other edges which are present in \(T\) are also present in \(T^{opt}\). Therefore, \(\text{dist}(T', T^{opt}) \leq \text{dist}(T', T^{opt}) - 1\), and hence, \(\gamma[y, x, a](T)\) is a distance improving floating \(\gamma\) transformation for this case.

\[ \square \]

Let \((x, y) \in T\) be a gap. Transformations \(\gamma\) on \((x, y)\) is the set of all transformations of the form \(\gamma[y, x, a]\) for all values of \(a\) such that \(\gamma[y, x, a]\) is a floating \(\gamma\)
Figure 2.7: Case 2 in the proof of Proposition 2.4.6. The bold edges are those known to be in $T^{opt}$, and the dashed edges are those known to not be in $T^{opt}$.

transformation. Similarly, \textit{transformations $\psi$ on $(x, y)$} is the set of all transformations of the form $\psi(y, x, a, c)$ for all values of $a$ and $c$ such that $\psi(y, x, a, c)$ is a floating $\psi$ transformation. Using Proposition 2.4.6, a BIS for any non-optimal tour is prescribed below.

**Definition 2.4.7. $\gamma$–$\psi$ BIS.**

Suppose $T$ is a non-optimal tour. Without loss of generality, there exists a gap $(x, y) \in T$. Then the union of transformations $\gamma$ on $(x, y)$ and transformations $\psi$ on $(x, y)$ constitutes a BIS of $T$. We call this set the $\gamma$–$\psi$ BIS.

The number of floating $\gamma$ transformations in the $\gamma$–$\psi$ BIS defined by the gap $(x, y) \in T$ is bounded by the number of choices for $a$ in $\gamma[y, x, a]$, which is bounded by $d$, the maximum degree of the graph. The number of opening $\psi$ transformations is bounded by $nd$, because there are up to $d$ choices for node $a$ in $\psi(y, x, a, c)$, and any node on the $(y, x, a)$ segment may be selected as a choice for $c$. Hence, the $\gamma$–$\psi$ BIS only contains $O(nd)$ transformations irrespective of the structure of the graph beyond $n$ and $d$. Obviously this is a significant improvement on the $O(n!)$ bound for the BIS consisting of all $k$-exchanges. A potential topic for future research is to investigate whether the bound of $O(nd)$ transformations may be improved further.
2.5 Principle 2: Restricting the Use of Opening Transformations

While a transformation that increases the number of gaps may decrease the distance from an optimal tour, it will always worsen the value of the tour by definition. Moreover, it is not guaranteed that an opening $\psi$ transformation decreases the distance to the optimal tour, and until we obtain the Hamiltonian cycle, there is no way of measuring whether an opening $\psi$ transformation has been effective. An opening transformation is like shooting in the dark without knowing exactly where the target is. Therefore, it is intuitive to only use this type of transformation as a last resort, when we have exhausted all other alternatives in the BIS of the tour.

Consider the $\gamma-\psi$ BIS defined by floating $\gamma$ and opening $\psi$ transformations defined on a gap $(x, y)$, as described in Definition 2.4.7. As discussed previously, there are at most $d$ choices for the node $a$ in a floating $\gamma[y, x, a]$ transformation where $d$ is the degree of $x$. For a sparse graph, it is possible to search this space very efficiently. In contrast, the number of opening $\psi$ transformations defined on a gap $(x, y)$ is bounded by $dn$. Here we discuss an alternative BIS prescription where the usage of opening $\psi$ transformations is more restricted than the $\gamma-\psi$ BIS.

Definition 2.5.1. Opening $\kappa$ transformation

Suppose $\kappa[y, x, a, c, d](T)$ is a valid transformation (maps tour $T$ to another valid tour). Then if $(y, x)$ is a gap and $(x, a), (b, c) \in E$, transformation $\kappa[y, x, a, c, d](T)$ is called an opening $\kappa$ transformation.

Note that the opening $\kappa$ is a composition of an opening $\psi$ transformation and a floating $\gamma$ transformation. In particular, $\kappa[y, x, a, b, c](T) = \gamma[y, b, c] \circ \psi[y, x, a, b](T)$. 
The following proposition was originally proved in [10]. However, we present a more concise proof here.

**Proposition 2.5.2.** Consider a non-optimal tour $T$. Let $(x,y) \in T$ and $(x,y) \notin T^{opt}$. Then at least one of the following is correct.

(i) $\exists a$ such that $\text{dist}(T', T^{opt}) < \text{dist}(T, T^{opt})$ for $T' = \gamma[y, x, a](T)$, where $\gamma[y, x, a](T)$ is a floating $\gamma$ transformation.

(ii) $\exists a, b, c$ such that $\text{dist}(T', T^{opt}) < \text{dist}(T, T^{opt})$ for $T' = \kappa[y, x, a, b, c](T)$, where $\kappa[y, x, a, b, c](T)$ is an opening $\kappa$ transformation.

(iii) $\exists a, b, c$ where $(a, e), (b, d) \in T$ such that $\text{dist}(T', T^{opt}) < \text{dist}(T, T^{opt})$ for $T' = \gamma(e, d, b) \circ \kappa[y, x, a, b, c](T)$ where $\kappa[y, x, a, b, c]$ is an opening $\kappa$ and $\gamma(e, d, b)$ is a floating $\gamma$ transformation.

**Proof.** Similar to the proof of Proposition 2.4.6, there exist a node $a$ such that $(x,a) \in T^{opt}$ but $(x,a) \notin T$. Let $h$ be the node that is adjacent to $a$ on $T$ on the $(y,x,a)$ segment, and let $e$ be the other node adjacent to $a$ on $T$.

We will consider three cases. For the first two cases, suppose $(a,h) \in T^{opt}$. Then, there is no edge of $T^{opt}$ going from $a$ to the $(y,x,a)^c$ segment. Because $T^{opt}$ does...
not contain sub-tours, there must be an edge with one endpoint on the \((y, x, a)\) segment and another endpoint on the \((y, x, a)^c\) segment. In particular, there exists a node \(b\) on the \((y, x, a)\) segment and a node \(c\) on the \((y, x, a)^c\) segment such that \((b, c) \in T^{opt}\). On \(T\), let \(d\) be the adjacent node to \(b\) on the \((y, x, b)\) segment, and let \(r\) be the adjacent node to \(b\) on the \((y, x, b)^c\) segment. Also let \(f\) be adjacent to \(c\) on the \((a, e, c)\) segment of \(T\).

**Case 1:** Suppose \((a, h) \in T^{opt}\) and \((b, d) \notin T^{opt}\). Transformation \(\varphi[y, x, a, b, c](T)\) removes the edges \(X = \{(y, x), (a, e), (d, b), (c, f)\}\) and adds the edges \(Y = \{(x, a), (e, d), (b, c), (f, y)\}\). In \(X\), at least three edges \((y, x), (a, e), (b, d) \notin T^{opt}\) are removed and at least two edges \((x, a), (b, c) \in T^{opt}\) are added. Thus, \(\text{dist}(T', T^{opt}) \leq \text{dist}(T, T^{opt}) - 1\). It follows that \(\varphi[y, x, a, b, c](T)\) is a distance improving transformation in case 1. This case is illustrated in Figure 2.9.

![Figure 2.9](image)

**Figure 2.9:** Case 1 in the proof of Proposition 2.5.2. The bold edges are those known to be in \(T^{opt}\), and the dashed edges are those known to not be in \(T^{opt}\).

**Case 2:** Suppose \((a, h) \in T^{opt}\) and \((b, d) \in T^{opt}\). Transformation \(\gamma(e, d, b) \circ \varphi[y, x, a, b, c](T)\) removes the edges \(X = \{(y, x), (a, e), (r, b), (c, f)\}\) and adds the edges \(Y = \{(x, a), (e, r), (b, c), (f, y)\}\). In \(X\), at least three edges \((y, x), (a, e), (b, r) \notin T^{opt}\) are removed and at least two edges \((x, a), (b, c) \in T^{opt}\) are added. Therefore, \(\text{dist}(T', T^{opt}) \leq \text{dist}(T, T^{opt}) - 1\) and \(\gamma(e, d, b) \circ \varphi[y, x, a, b, c](T)\) is a distance improving transformation in case 2. Figure 2.10 illustrates this case.

**Case 3:** Suppose \((a, h) \notin T^{opt}\). Then \(\gamma[y, x, a](T)\) transforms \(T\) to \(T' = (T \setminus \)
Figure 2.10: Case 2 in the proof of Proposition 2.5.2. The bold edges are those known to be in $T^{opt}$, and the dashed edges are those known to not be in $T^{opt}$.

Let $X \cup Y$ for $X = \{(x, y), (a, h)\}$ and $Y = \{(x, a), (h, y)\}$. We know that the two edges $(x, y), (a, h) \notin T^{opt}$ and at least one edge $(x, a) \in T^{opt}$. Therefore, $\text{dist}(T', T^{opt}) \leq \text{dist}(T, T^{opt}) - 1$. Then $\gamma[y, x, a](T)$ is a distance improving opening $\psi$ transformation in case 3. This case is illustrated in Figure 2.11. □

Figure 2.11: Case 3 in the proof of Proposition 2.5.2. The bold edges are those known to be in $T^{opt}$, and the dashed edges are those known to not be in $T^{opt}$.

Proposition 2.5.2 proves that the compositions of $\gamma$ and $\kappa$ are sufficient for transforming the tour to the optimal tour. Furthermore, transformation $\kappa$ is restricted in the use of opening transformations by comparison to $\psi$ transformations, because the opening $\kappa$ transformation is a composition of an opening transformation $\psi$ followed by a fixed floating $\gamma$ transformation. In other words, for any tour, the opening $\psi$ transformations are restricted by the condition that the particular ensuing floating $\gamma$ transformation must also be available. In particular, $\kappa[y, x, a, b, c]$
ensures that an opening $\psi(y, x, a, c)$ transformation is performed only if there exists an edge $(c, d)$ where $d$ is on the $(x, y)$ segment. As a consequence, the combination of floating $\gamma$ and opening $\kappa$ is a superior option for prescribing a BIS, compared to the combination of floating $\gamma$ and opening $\psi$ in its unrestricted form.

Using Proposition 2.5.2, we may construct the following BIS for any non-optimal tour.

**Definition 2.5.3. $\gamma$–$\kappa$ BIS**

Suppose $T$ is a non-optimal tour. Without loss of generality, there exists a gap $(x, y) \in T$. Then the union of the following sets of transformations constitutes a BIS for the tour.

- Floating $\gamma[y, x, a]$ for any feasible choice of $a$.
- Opening $\kappa$ transformations of the form $\kappa[y, x, a, b, c]$ for possible choices of $a, b$ and $c$.
- Transformations of the form $\gamma[e, d, b] \circ \kappa[y, x, a, b, c](T)$ for possible choices of $a, b, c, e, d$ and $b$ where the $\kappa[y, x, a, b, c](T)$ transformation is an opening $\kappa$ transformation and $\gamma(e, d, b)$ is a possible $\gamma$ transformation.

The set of transformations defined above is called the $\gamma$–$\kappa$ BIS.

## 2.6 Principle 3: Prioritizing Transformations with Lower Computational Cost

To obtain an optimal tour, we start from some initial tour $T_{init}$. After a series of transformations, we arrive at the current tour $T$. Given any current tour $T$, the key question is how to select the next transformation from the BIS of the
current tour? Let an *available transformation* be a mapping from $T$ to another tour $T'$ where $T'$ has not been visited previously. Note that in order to select a transformation in a BIS, we are not required to search the entire space. Instead, to save computational time, we can perform the first available transformation that the algorithm identifies. This significantly reduces the average time spent on finding a transformation. This strategy is also significantly enhanced if we provide a relative priority of the transformations in the BIS which determines where the algorithm should look first. Previously, we have argued that the opening transformations must have the lowest priority (Principle 2), and so we adopt that choice here. Next, we develop concepts and instructions for ranking the other transformations.

It is possible, and often desirable, to limit the selection of opening transformations further by expanding the $\gamma \rightarrow \kappa$ BIS. For example, consider the $\gamma \rightarrow \kappa$ BIS on a tour $T$ with a gap $(x, y)$. If there are no available floating $\gamma$ transformations on $(x, y)$, then the only option is to perform an opening $\kappa$ transformation. However, it might be the case that a floating transformation not contained in the BIS, would be available on $T$. Therefore, it may be desirable to identify “shortcut” floating and closing transformations which are useful often enough to be worth adding to the BIS, so that opening transformations are required less often. Obviously, the use of these should be limited to ensure that the BIS does not grow too large.

In the $\gamma \rightarrow \kappa$ BIS, there are at most $O(nd^2)$ transformations. In our implementation, we implement certain shortcut transformations, but take care to avoid any shortcuts which expand the BIS by more than $O(nd^2)$ transformations for a given gap on a tour. In particular, the extended $\gamma \rightarrow \kappa$ BIS includes the following transformations, given here in order of priority:

1. Closing transformation of the form $\gamma \circ \gamma$ or $\psi$.

2. Sequential floating $k$–exchanges for $k \leq 5$. 

39
(3) Non-sequential floating $k$-exchanges for $k \leq 4$.

(4) Opening transformations of the $\gamma$-$\kappa$ BIS.

Note that the floating $\gamma$ transformations are contained in item (2); therefore the extended extended $\gamma$-$\kappa$ BIS defined above is a superset of the $\gamma$-$\kappa$ BIS. It is worth mentioning that it is possible to improve the speed of the algorithm by expanding item 1 to the Lin-Kernighan type sequential exchanges. However, we have observed experimentally that item (1) does not affect the effectiveness of the algorithm in obtaining optimal tours beyond slightly improving the speed of the algorithm in some instances. To differentiate our approach from the Lin-Kernighan type algorithms, we have chosen only to implement very simple 3-opt transformations. In fact, in our experience, even the removal of item (1) transformations will not significantly alter the effectiveness of the algorithm in most instances.

2.7 The Basic Algorithm

We now present Algorithm 1, which outlines the basic design of the BIS-based HCP algorithm based on the ideas presented in this chapter. In contrast to SLH presented in [10], Algorithm 1 is not a heuristic algorithm. To create a heuristic algorithm, one can place an artificial limit for the maximum number of visited tours. For random graphs, an $n^2$ bound is usually sufficient. However, in our experience, choosing an $n^3$ bound is a better choice because it is effective for solving almost all graphs, even when there are very few Hamiltonian cycles available.

**Proposition 2.7.1.** Algorithm 1 is an exact algorithm. In other words, Algorithm 1 always finds a Hamiltonian cycle for a Hamiltonian Graph, or returns the tour with the least gaps for a non-Hamiltonian graph.

**Proof.** Let $T^{opt}$ be an optimal tour of the graph. Suppose $T$ is the tour in $VisitedTours$ where $\text{dist}(T, T^{opt})$ is minimal amongst all tours in $VisitedTours$
Algorithm 1: BIS-based HCP algorithm

(1) Initialization:
   
   (a) A starting initial tour $T_{\text{init}}$. $T_{\text{init}}$ could be any permutation of nodes, obtained randomly or given as an initial input. Record the minimum number of gaps $\text{MinGaps} = g(T_{\text{init}})$.
   
   (b) An ordered list of fingerprints\(^1\) of tours denoted by $\text{VisitedTours}$. This list is empty at the start.
   
   (c) A list of transformations $\text{TransformationsList}$ for the purpose of backtracking when necessary. The list is initially empty.
   
   (d) Current tour $T = T_{\text{init}}$ and best tour $T_{\text{best}} = T_{\text{init}}$.

(2) Search and find the first available transformation in the BIS of $T$ to a tour $T'$. Note that an available transformation maps $T$ to a tour that is not already present in the $\text{TransformationsList}$.

(3) If an available transformation is found:
   
   (a) If $\text{MinGaps} > g(T')$, empty the $\text{TransformationsList}$ and $\text{VisitedTours}$, set $\text{MinGaps} = g(T')$ and set $T_{\text{best}} = T'$. Otherwise, add the last transformation to the end of the $\text{TransformationsList}$ and add $T$ to the $\text{VisitedTours}$ list.
   
   (b) If $\text{MinGaps} = 0$, output $T_{\text{best}}$ and stop.
   
   (c) Set $T = T'$. Return to step (2).

(4) If no available transformation is found in the BIS and $\text{TransformationsList}$ is not empty, backtrack by performing the inverse of the last transformation in the $\text{TransformationsList}$. Set $T$ to be the obtained tour and remove the last element of the $\text{TransformationsList}$. Return to step (2).

(5) If there are no more element in $\text{TransformationsList}$, there are no Hamiltonian cycles in the graph. Output the $T_{\text{best}}$ with $\text{MinGaps}$ as the optimal solution and declare the graph Non-Hamiltonian.
and \( g(T) \neq 0 \). Since transformations are selected from a BIS, there exists a transformation of \( T \) to another tour \( T' \) where \( \text{dist}(T', T^{\text{opt}}) < \text{dist}(T, T^{\text{opt}}) \). Furthermore, this transformation is an available transformation, otherwise \( \text{dist}(T, T^{\text{opt}}) \) is not minimal amongst all tours in the \textit{VisitedTours}. Therefore, there are three scenarios:

- Eventually, a sequence of transformations leads to a tour with a reduced distance to \( T^{\text{opt}} \).
- Eventually, a sequence of transformations leads to a Hamiltonian cycle that is not equal to \( T^{\text{opt}} \).
- Eventually, the backtracking feature will lead to transforming \( T \) to \( T' \).

In all cases, either an optimal tour is obtained or the distance to the optimal tour will decrease. Therefore, an optimal tour will always be found. \( \square \)

There is an enormous discrepancy between the best known bound for the worst-case complexity of Algorithm 1 and its actual performance on various instances of HCP. The algorithm has been successful in solving various instances of HCP, obtaining performance that is on-par with the state-of-the-art solvers. As mentioned before, even structurally difficult instances, where there are only a few Hamiltonian cycles, are often solved within \( n^3 \) transformations. The only exception that in the literature that we are aware of are the Fleishner graphs [40], where it usually necessary to perform up to \( n^4 \) iterations. On the other hand, the best known worst-case complexity of Algorithm 1 is \( O(n!) \). We note that this discrepancy between the worst-case complexity and actual performance is not unique to this algorithm. For example, the worst case complexity of Concorde, which is widely regarded to be the best exact TSP solver, is also \( O(n!) \) while its actual performance is typically within polynomial bounds.

The competitive performance, despite the combinatorial explosion of the worst-
case scenario, seems to be due to the intelligent management of computational resources in the method. Principles 2 and 3 ensure that computational resources are first expended in search spaces that require the least computational resources. By deprecimating computationally expensive transformations, we initially search through the tours that are fastest to obtain. The average cost of performing transformations is very low at the start. Of course, as the algorithm progresses and improvement is not found, the average computational cost of each transformation increases. However, by prioritizing the search in the spaces that require the least computational resources, we avoid most of the search space while providing very good experimental odds of obtaining an optimal tour in most instances.

The effectiveness of the algorithmic method is first demonstrated in [10], and is discussed in the next chapter. In particular, Chapter 3 includes an extensive computational study, benchmarking an implementation of Algorithm 1 against other state-of-the-art solvers. The computational study primarily focuses on difficult instances of HCP, which we have found in the literature, or have generated through various methods.

### 2.7.1 Software

I have implemented a Java prototype of the concepts developed in this Section. The full-featured version\(^2\) of the SLH 2.49 is implemented in C++. SLH 2.49 is a heuristic implementation with a bound of maximum \(n^3\) transformations. The Windows and Linux versions of SLH 2.49 can be downloaded online at https://www.travelingsalesman.org.

\(^2\)The implementation of the full-featured version of SLH is not part of this dissertation.
Chapter 3

Benchmarking Traveling Salesman Problem Algorithms by Difficult Sparse Instances

In this chapter, I discuss the creation of a new benchmark set of difficult HCP instances which are used to evaluate the performance of TSP algorithms. This study is motivated by three primary reasons. First, the chapter contains a thorough study of the performance of SLH v2.49 (i.e., SLH2), an implementation of Algorithm 1 introduced in the previous chapter, on various examples compared to several other state-of-the-art TSP algorithms. Second, we introduce a new perspective on selecting benchmark examples that provide important insights that are different and complementary to the insights obtained from the existing HCP and TSP benchmark sets. Finally, the computational study carried out in this chapter has played an important role in selecting the algorithmic approaches discussed later in Chapter 4.
3.1 Introduction

When analyzing the performance of algorithms or developing a new algorithm, it is essential to have good benchmark instances. Benchmark instances are useful for two primary reasons. First, they allow for comparisons with competing algorithms. Second, they stress algorithms in various ways, which helps to identify weaknesses or even potential bugs in the implementation.

Given their importance, it is natural to ask what constitutes a good benchmark set. It seems sensible that a good benchmark set should contain sufficient variety to indicate the expected performance of the algorithm in general; that is, on instances not contained in the benchmark set. Therefore, a good benchmark set should provide insight on the following:

- **Generic performance:** What performance can be expected from the algorithm on a generic instance.
- **Weaknesses of an algorithm:** The situations in which the performance of an algorithm deteriorates.
- **Comparative advantages:** When comparing two algorithms, the situations in which the performance of the two algorithms diverges.

One common approach for generating benchmark sets is to randomly generate generic instances through sampling the instance space. Well known examples such benchmark sets include SATLIB [107] for boolean satisfiability, and TSPLIB [104] for TSP and its variations. This approach is good at identifying which algorithms perform best on random instances, and which implementations of those algorithms are the fastest. However, it has been recognized that this approach often provides little insight into the weaknesses of an algorithm, or the comparative advantages of competing algorithms [67].
Often, algorithms exhibit remarkably different performance on randomly generated instances than they do on instances which arise from applications. Typically, the latter contains special structures that are extremely rare in random instances. Such special structures sometimes impact the performance of an algorithm significantly. Consequently, benchmarking the performance of an algorithm on randomly generated instances may give little or no insight to the weaknesses of an algorithm for solving structured instances. These algorithmic weaknesses can only be exposed by a more comprehensive benchmark set that focuses on structural difficulties. In addition, we have observed that the algorithms which perform well on structurally difficult instances also tend to perform well on randomly generated instances. For these reasons, creating benchmark sets with difficult structures could lead to unique insights about comparative performance of algorithms.

In recent times, there has been growing interest in creating benchmark instances by sampling particular sections of the instance space where the performance of different algorithms diverges. This approach first attempts to characterize the instance space according to the features of the instances (ideally features which can be linked to difficulty), and then identifies which algorithms perform best in different sections of the instance space. If a meaningful set of features is chosen to characterize the instance space, this approach should permit the identification of comparative advantages in competing algorithms. Examples of these studies that rely on machine learning and statistics include Leyton-Brown et al. [89] for winner determination problem, Smith-Miles et al. [116] for Traveling salesman problem, Cho et al. [26] and Hall and Posner [54] for knapsack problems, and Smith-Miles et al. [114] for job shop scheduling problem.

The approach which we advocate here is to construct inherently difficult examples with minimal size. That is, small examples that are difficult for many competing algorithmic approaches. In a sense, this approach focuses on finding or constructing **challenging outlier instances** with the highest level of difficulty for the most
competent approaches. This approach is best at exposing the situations in which an algorithm performs poorly. Indeed, we have found these insights extremely helpful for identifying how to improve our own algorithms in their developmental stages. Furthermore, studying challenging outlier examples can provide clues on which features may be the cause of the difficulty in these examples.

Of course, researchers have long attempted to construct such challenging outlier instances for various algorithms. Perhaps the most famous example is the seminal paper by Selman et al [109] in which they describe the ideal ratio of clauses to variables in difficult randomly generated instances of boolean satisfiability. However, for many other difficult and widely studied problems, similar results are rare or even non-existent. In particular, literature does not currently contain benchmark sets for the Traveling salesman problem (TSP) where small challenging outlier examples are explicitly collected and systematically studied as a benchmark set.

In this chapter, I focus on TSP, along with the closely related Hamiltonian cycle problem (HCP). Despite the latter also being a widely studied problem, there are practically no recognized benchmark sets for HCP, and the few benchmark instances which do exist are trivial to solve for modern HCP algorithms and hence provide no real insight about their relative performance. We will consider a particular type of benchmark instance, which can be viewed as either an instance of HCP or TSP, and which has a very different character to instances in the existing benchmark sets for TSP. We will show that instances of this type stress the most powerful TSP algorithms in ways that the existing benchmark sets do not, and in doing so expose areas of potential improvement in those algorithms.

The benchmark set described in this chapter can be downloaded online at http://fhcp.edu.au/tsphcp [11].
3.1.1 TSP Benchmarks

The effectiveness of a TSP heuristic on a given instance relies on many factors such as the size and structure of the instance. Some heuristics may be quite effective at solving a certain class of problems, but perform poorly on a different class of problems. Hence, to fully evaluate the effectiveness of a TSP heuristic, it should be tested on a wide range of instances incorporating various different characteristics. For this purpose, many benchmark sets have been developed. Perhaps the most famous TSP benchmark instance is the 49 city problem (corresponding to capital cities of mainland USA states), which was first solved to provable optimality by Dantzig et al. [33] in the 1950s. More recently, benchmarking sets such as TSPLIB [104], the World TSP challenge [125], and the DIMACS Challenge set [71] have been developed and are used by developers of TSP algorithms to evaluate their performance. TSPLIB is a set of randomly generated instances of TSP (and its variations) containing up to 85,900 cities. The World TSP challenge contains a number of 2D-TSP instances\footnote{1} based on satellite data of cities in various countries throughout the world. The DIMACS TSP challenge, first introduced in 2000, includes a set of TSP instances of size up to 10 million cities, consisting primarily of 2D-TSP instances, as well as seven randomly generated instances. When it was first released, DIMACS invited developers of state of the art algorithms to submit their best solutions to these problems. For many of these instances, it is still unknown whether the best known tours are optimal.

All three of these benchmark sets are similar in character, containing either randomly generated instances, or 2D-TSP instances. The randomly generated instances tend to be difficult only because of large size. The 2D-TSP instances are sometimes difficult even at moderately small size; when this is the case, it is usually because the cities are clustered. Indeed, it seems that truly difficult instances

\footnote{1}{A 2D-TSP instance is a TSP where the cities are embedded on a plane and the distances to be computed as Euclidean distances.}
require the presence of particular structures.

In this chapter, we will consider TSP instances which have entirely different character to those in the recognized benchmark sets. Our approach involves considering the Hamiltonian cycle problem (HCP). Recall from Definition 1.4.1 that HCP may be stated as follows: Given a graph $G$, determine whether there exists a simple cycle that includes all vertices in the graph. HCP is known to be \textit{NP}-complete [75], and therefore it encompasses some of the significant difficulty of TSP. There are many ways to convert an instance of HCP to an instance of TSP. However, a simple conversion from HCP to TSP is as follows: set the distance between any two cities $i$ and $j$ equal to 0 if the edge $(i, j)$ exists in $G$, and equal to 1 otherwise. Then, any tour with total distance equal to zero must be optimal, and will correspond to a Hamiltonian cycle in $G$. Therefore we can think of HCP as the binary form of TSP.

Since HCP is equivalent to binary TSP, we can use it to induce graph structure that is very difficult (often impossible) to include in 2D-TSP instances, and extremely unlikely to occur in randomly generated instances. This in turn provides an opportunity to create new benchmark instances that will challenge TSP solvers in ways other benchmark sets will not. There are two additional benefits to generating benchmark problems in this way; first, if the underlying instance of HCP is known to be Hamiltonian, then we know in advance the length of an optimal tour, so we can easily confirm whether a heuristic has been successful in finding it. And second, producing a benchmark set for TSP in this way has the added benefit of also providing a benchmark set for HCP. As mentioned previously, there are very few recognized benchmark sets for HCP, and those few which exist (most famously, the nine HCP instances contained in TSPLIB [104]) contain very few instances, of which all are trivial to solve for modern HCP algorithms (see [62, 10]).

In this chapter we compile a list of good candidate HCP instances to be used as
TSP benchmark instances. We also include an extensive benchmarking exercise which has taken eight years of CPU time to be completed. In Section 3.2 we report on the best HCP instances from literature for this purpose. In Section 3.3 we consider randomly generated HCP instances, and propose a method for iteratively modifying random instances to obtain new instances that may challenge TSP solvers. In Section 3.4, we consider the possibility of converting challenging instances of other $NP$-complete problems into instances of HCP. Each section includes a report on our computational study. Finally, in Section 3.5 we conclude with an analysis of the results obtained from the benchmarking exercise.

3.1.2 Properties that Contribute to the Difficulty of Instances

As mentioned above, the performance of TSP heuristics on instances depends on a number of factors. When comparing two heuristic methods for TSP, one may outperform the other on one class of instances while falling behind on a different class. Obviously, one such factor that contribute to difficulty is size; larger instances are generally more difficult to solve. The difficulty is exacerbated by the additional memory requirements when solving on a standard computer. Most existing benchmark sets focus on the size of the problem as the primary measure of difficulty. For this reason, we have instead put our focus on other factors that contribute to the difficulty of the problem. Since our instances are all based on an underlying instance of HCP (that is, a graph), we loosely refer to these factors as graph structure.

We claim that the performance of solvers on certain instances will vary given the presence or absence of certain graph structures. For example, some TSP algorithms can take advantage of symmetry when it is present. Among other graph features that may affect performance of an algorithm are density of edges, repeated structures, and prevalence of Hamiltonian cycles. At this stage, the exact
effect of certain graph structures on the performance of TSP algorithms is not well understood.

As indicated already, in the present work we focus on considering difficult instances with minimal size. We believe that such an approach is potentially more rewarding in terms of identifying weaknesses and strengths of an algorithm in a broader context. In most cases, the instances considered are members of an infinite family, and so larger instances can be constructed if so desired.

### 3.1.3 Algorithms to be Tested

In order to determine which of our collected instances provided the greatest challenge for algorithms, we conducted a benchmarking exercise. In that exercise, we used the following four algorithms, including three TSP algorithms, and one specialized HCP heuristic.

- **Chained Lin-Kernighan algorithm (v 3.12.19) (CLK)** [7] is a recent implementation of the famous Lin-Kernighan heuristic algorithm [88], and is included in the Concorde package.

- **Helgaun’s implementation of Lin-Kernighan (v 2.0.7) (LKH)** [62] is a recent implementation of the famous Lin-Kernighan heuristic known for its success of at discovering the best known solutions for many benchmark examples including the World TSP challenge. It holds a number of records for TSP.

- **Snakes and Ladders Heuristic (v 2.49) (SLH2)** [10] is an efficient heuristic algorithm designed specifically for solving HCP instances. It is described in detail in Chapter 2.

- **Concorde (v 3.12.19)** [7] is arguably the most well-known exact TSP solver. Though Concorde’s worst-case complexity is combinatorial, in practice, so-
olutions are often obtained within polynomial time bounds because it uses clever linear programming techniques to obtain and verify an optimal solution. Thus, it is reasonable to compare this solver with heuristic approaches when a time-limit is deployed for the processing time.

Of course, for many of the instances we considered when creating the benchmark set, we found that all four algorithms were able to solve them trivially. Hence, in the sections that follow, we only include instances (or families of instances) that stressed at least one of the algorithms. In keeping with our desire to consider small instances, all instances considered contain fewer than 10,000 cities, other than a sole exception coming in Section 3.3.

We performed all of the tests on a single core of an AMD Opteron(tm) Processor 6282 SE on a UNIX OS server. We have limited our usage of virtual memory and computational time to 4 GBs and 24 hours respectively for a single run. The execution time reported is the average time for one successful run of a solver, only averaging over the runs where the optimal tour is obtained. In Tables 3.3–3.12, a test instance is counted as unsolved for four different reasons described in Table 3.1.

Table 3.1: The four situation where a test instance is counted as unsolved. In Tables 3.3–3.12, the relevant situation is marked by the sign indicated in the Marked column.

<table>
<thead>
<tr>
<th>Situation</th>
<th>Marked</th>
</tr>
</thead>
<tbody>
<tr>
<td>The solved has concluded its process and an optimal tour is not found at the end of the run.</td>
<td></td>
</tr>
<tr>
<td>The solver could not find an optimal tour within 24 hours.</td>
<td>*</td>
</tr>
<tr>
<td>The solver requested more than 4 GBs of virtual memory.</td>
<td>**</td>
</tr>
<tr>
<td>The solver halted unexpectedly and reported a bug.</td>
<td>***</td>
</tr>
</tbody>
</table>
3.2 HCP Instances in the Literature

In this section, we summarize some of the most difficult HCP instances that are known in literature, and attempt to solve them using the four test algorithms. It should be noted that these instances were not initially designed to be difficult for HCP heuristics, however their particular structures or features lend themselves to difficulty. For each instance, we produced 100 random relabellings of the vertices, and ran the four algorithms on each of the 100 relabelled instances. We report on the number of times an optimal tour was obtained, as well as the average time it took for the algorithm to run in the cases where it was successful.

3.2.1 Generalized Peterson Graphs

Description: Generalized Peterson graphs $GP(p, k)$ are a family of 3-regular graphs which are constructed by combining an inner star polygon and an outer regular polygon. Graph $GP(p, k)$ contains $n = 2p$ nodes and $m = 3p$ edges. A precise description is given in [122]. Certain choices of $n$ and $k$ result in graphs with special characteristics, in particular graphs with a high degree of symmetry and known number of Hamiltonian Cycles [108]. In Table 3.2 we summarize three classes that we focus on here.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
\text{n/2} & $k$ & Number of Hamiltonian Cycles \\
\hline
$n/2 \equiv 1 \pmod{6}$ & 2 & \text{n} \\
$n/2 \equiv 3 \pmod{6}$ & 2 & 3 \\
$n/2 \equiv 5 \pmod{6}$ & 2 & 0 \\
\hline
\end{tabular}
\caption{Three types of Generalized Peterson graphs $GP(n/2, k)$ and their respective number of Hamiltonian cycles. Time reported in seconds and the Solved columns report the number of times an optimal tour was obtained out of 100 runs.}
\end{table}

Note that for $n \equiv 5 \pmod{6}$ and $k = 2$, the graphs are non-Hamiltonian. Furthermore, it is known that the addition of any edge will introduce Hamiltonian cycles [19] (hypohamiltonian property). Hence, in these cases, we add one random
Table 3.3: Performance of the four algorithms on generalized Peterson graphs. Time reported in seconds and the Solved columns report the number of times an optimal tour was obtained out of 100 runs.

<table>
<thead>
<tr>
<th>Name</th>
<th>n</th>
<th>Concorde</th>
<th>CLK</th>
<th>LKH</th>
<th>SLH2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Solved</td>
<td>Time</td>
<td>Solved</td>
<td>Time</td>
</tr>
<tr>
<td>GPN</td>
<td>122</td>
<td>0*</td>
<td>NA</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>GPN</td>
<td>244</td>
<td>100</td>
<td>0.69</td>
<td>100</td>
<td>0.02</td>
</tr>
<tr>
<td>GPN</td>
<td>482</td>
<td>0*</td>
<td>NA</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>GPN</td>
<td>998</td>
<td>0*</td>
<td>NA</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>GP3</td>
<td>126</td>
<td>0*</td>
<td>NA</td>
<td>1</td>
<td>0.09</td>
</tr>
<tr>
<td>GP3</td>
<td>246</td>
<td>0*</td>
<td>NA</td>
<td>0</td>
<td>0.22</td>
</tr>
<tr>
<td>GP3</td>
<td>486</td>
<td>0*</td>
<td>NA</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>GP3</td>
<td>1002</td>
<td>0</td>
<td>NA</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>GP0</td>
<td>130</td>
<td>100</td>
<td>0.24</td>
<td>100</td>
<td>0.07</td>
</tr>
<tr>
<td>GP0</td>
<td>250</td>
<td>98***</td>
<td>223.11</td>
<td>36</td>
<td>0.20</td>
</tr>
<tr>
<td>GP0</td>
<td>490</td>
<td>0*</td>
<td>NA</td>
<td>21</td>
<td>0.50</td>
</tr>
<tr>
<td>GP0</td>
<td>1006</td>
<td>0*</td>
<td>NA</td>
<td>7</td>
<td>1.32</td>
</tr>
</tbody>
</table>

edge to create Hamiltonian graphs for benchmarking purposes. We will refer to these three classes of Generalized Petersen graphs as GPN, GP3 and GP0 respectively.

Benchmarking notes: The classes of Generalized Petersen Graphs considered are highly symmetric, and GPN and GP3 instances contain relatively few Hamiltonian cycles. It is not clear how many Hamiltonian cycles the GP0 instances have once an edge is added, however all of them must include that edge. These characteristics make it very difficult for algorithms to take advantage of structures within the graph. Therefore they constitute very difficult examples for most algorithms even when the size of the graph is small, and it can be seen in Table 3.3 that all considered algorithms encountered some difficulty solving them, either in terms of unsuccessful runs, or rapidly increasing solving time. As a result, they constitute extremely useful instances.
3.2.2 Sheehan Graphs

Description: Sheehan [110] described a family of maximally dense uniquely Hamiltonian graphs; that is, Hamiltonian graphs with only a single Hamiltonian cycle, that contain the maximum possible ratio of edges to vertices.

Benchmarking notes: The high density of edges and the fact that the graphs contain only one Hamiltonian cycle make Sheehan graphs especially interesting instances for heuristics that struggle with high density of edges. However, these graphs contain a structural weakness that can be exploited; one at a time, edges can be identified as being impossible to include in a Hamiltonian cycle and removed, until only the Hamiltonian cycle remains. As can be seen in Table 3.4, Concorde and LKH, which both make use of linear programming techniques, performed well on this set, while CLK and SLH2 did not.

Table 3.4: Performance of the four algorithms on Sheehan Graphs. Time reported in seconds and the Solved columns report the number of times an optimal tour was obtained out of 100 runs.

<table>
<thead>
<tr>
<th>Name</th>
<th>n</th>
<th>Concorde</th>
<th>CLK</th>
<th>LKH</th>
<th>SLH2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Solved</td>
<td>Time</td>
<td>Solved</td>
<td>Time</td>
</tr>
<tr>
<td>SH_64</td>
<td>64</td>
<td>100</td>
<td>0.86</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>SH_125</td>
<td>250</td>
<td>100</td>
<td>3.56</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>SH_250</td>
<td>500</td>
<td>100</td>
<td>47.92</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>SH_500</td>
<td>1000</td>
<td>100</td>
<td>46.65</td>
<td>0</td>
<td>NA</td>
</tr>
</tbody>
</table>

3.2.3 Modified Flower Snarks

Description: A Snark is a connected, bridgeless 3-regular graph with chromatic index equal to 4. Most standard definitions also require that the graph have minimum girth 5 [103]. All Snarks are non-Hamiltonian, and many are hypohamiltonian. One infinite family of hypohamiltonian Snarks is the Flower Snarks discovered by Isaacs [70]. We consider a modification of Flower snarks obtained
by adding one random edge to the graph, such that the resulting graph is Hamiltonian. In particular, we first find a tour with one gap, which is easy to obtain by SLH2 or LKH, and then add an edge instead of the gap. Empirically, it appears that instances constructed in this way contain several Hamiltonian cycles. Nevertheless, the instances prove difficult for some TSP heuristics as the size increases.

Benchmarking notes: Most of the tested algorithms were capable of solving the modified Flower Snarks, but Concorde and CLK were starting to struggle once the number of vertices reached 1004. In contrast, SLH2 and LKH both found these instances easy to solve. The results are summarized in Table 3.5.

**Table 3.5:** Performance of the four algorithms on modified Flower Snarks. Time reported in seconds and the Solved columns report the number of times an optimal tour was obtained out of 100 runs.

<table>
<thead>
<tr>
<th>Name</th>
<th>n</th>
<th>Concorde</th>
<th>CLK</th>
<th>LKH</th>
<th>SLH2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Solved</td>
<td>Time</td>
<td>Solved</td>
<td>Time</td>
<td>Solved</td>
</tr>
<tr>
<td>SN_124</td>
<td>124</td>
<td>100 0.40</td>
<td>73 0.09</td>
<td>100 0.01</td>
<td>100 0.02</td>
</tr>
<tr>
<td>SN_252</td>
<td>252</td>
<td>100 0.40</td>
<td>100 0.22</td>
<td>100 0.02</td>
<td>100 0.01</td>
</tr>
<tr>
<td>SN_500</td>
<td>500</td>
<td>100 1.22</td>
<td>95 0.59</td>
<td>100 0.11</td>
<td>100 0.13</td>
</tr>
<tr>
<td>SN_1004</td>
<td>1004</td>
<td>98* 549.45</td>
<td>43 1.44</td>
<td>100 0.48</td>
<td>100 0.87</td>
</tr>
</tbody>
</table>

### 3.2.4 Fleischner Graphs

Description: Fleischner [40] introduced two families of graphs with minimum degree 4 and a unique Hamiltonian cycle, that are 2-connected and 3-connected respectively. The smallest of these graphs have 170 vertices\(^2\) and 408 vertices respectively, and larger instances are produced by chaining multiple copies of certain graphs together in a prescribed fashion. The larger instances have \(169k\) vertices and \(85 + 323k\) vertices for \(k = 2, 3, \ldots\), respectively. The minimum degree being 4 causes difficulty for some heuristics, as it significantly hampers the use of

\(^2\)Strictly speaking, the smallest Fleischner graph has 338 vertices, produced by joining two copies of a 169 vertex graph together. However, a 170-vertex graph is produced by replacing the second copy with a single vertex of degree 2, violating the degree 4 requirement but still producing a difficult instance.
propagation techniques for branch and bound type approaches.

Benchmark notes: As can be seen in Table 3.6, the Fleischner graphs posed an enormous challenge for all tested heuristics, with practically all attempts resulting in failure. These instances demonstrate that minimum degree, when paired with a low number of Hamiltonian cycles, should be considered a measure of difficulty. Indeed, this provides additional motivation to address the, still open, question posed by Fleischner [40] on whether any uniquely Hamiltonian graphs with minimum degree 5 or higher exist.

**Table 3.6: Performance of the four algorithms on Fleischner graphs. Time reported in seconds and the Solved columns report the number of times an optimal tour was obtained out of 100 runs.**

<table>
<thead>
<tr>
<th>Name</th>
<th>$n$</th>
<th>Concorde</th>
<th>CLK</th>
<th>LKH</th>
<th>SLH2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Solved</td>
<td>Solved</td>
<td>Time</td>
<td>Solved</td>
</tr>
<tr>
<td>FLS 170</td>
<td>170</td>
<td>0*</td>
<td>NA</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>FLS 338</td>
<td>338</td>
<td>0*</td>
<td>NA</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>FLS 507</td>
<td>507</td>
<td>0*</td>
<td>NA</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>FLS 676</td>
<td>676</td>
<td>0*</td>
<td>NA</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>FLS 845</td>
<td>845</td>
<td>0*</td>
<td>NA</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>FLS 1014</td>
<td>1014</td>
<td>0*</td>
<td>NA</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>FLS 408</td>
<td>408</td>
<td>0*</td>
<td>NA</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>FLS 731</td>
<td>731</td>
<td>0*</td>
<td>NA</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>FLS 1054</td>
<td>1054</td>
<td>0*</td>
<td>NA</td>
<td>0</td>
<td>NA</td>
</tr>
</tbody>
</table>

### 3.3 Randomly Generated Instances

It is widely known that randomly generated graphs do not typically provide difficult instances of HCP, and indeed, can usually be solved in almost linear time [45]. However, for completeness, we consider randomly generated instances here, as well as discussing a procedure for modifying randomly generated graphs to increase their difficulty for specific algorithms.
3.3.1 Randomly Generated HCP Instances

Description: There are various procedures for generating random graphs with any number of properties. Here, we use the standard method to produce random regular graphs described by Wormald [126], and in particular, we generate 3-regular graphs \(^3\). Indeed, HCP restricted to 3-regular graphs is known to be \(\mathcal{NP}\)-complete [37, 46]. It could be argued that performing tests on sets of random 3-regular graphs provides a good indicator of how an algorithm might be expected to perform on a “typical” 3-regular graph. However, in practice real-world problems display different properties rarely captured by random graphs. As long as a few conditions in terms of edge density and connectivity are met, random graphs are almost always Hamiltonian. This includes random regular graphs, which are known to almost always be Hamiltonian [106] and typically contain many Hamiltonian cycles. Hence, we expect good heuristics to perform well on these graphs, with difficulty only arising as a result of size.

Benchmarking notes: In Table 3.7, the reported time is the average time over 2000 samples for each algorithm and each size. All reported failures indicate the inability of the process to find an optimal tour at the conclusion of the process. As can be seen, all algorithms performed well on these instances. For this reason, we have not included these instances in our benchmark repository [11].

3.3.2 Modifying Random Instances

Description: As seen above, randomly generated Hamiltonian graphs are typically unchallenging for competent algorithms. However, it is possible to intelligently modify these graphs to produce more difficult instances of arbitrarily large order. In particular, it is possible to create instances which most likely contain very few

\(^3\)It is known that almost all random regular graphs are Hamiltonian [106].
Table 3.7: Performance of the four algorithms on randomly generated HCP instances. Time reported in seconds and the Solved columns report the number of times an optimal tour was obtained out of 100 runs.

<table>
<thead>
<tr>
<th>Size</th>
<th>Sample</th>
<th>Concorde</th>
<th>CLK</th>
<th>LKH</th>
<th>SLH2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Solved</td>
<td>Time</td>
<td>Solved</td>
<td>Time</td>
</tr>
<tr>
<td>250</td>
<td>2000</td>
<td>100</td>
<td>1.46</td>
<td>100</td>
<td>0.78</td>
</tr>
<tr>
<td>500</td>
<td>2000</td>
<td>100</td>
<td>3.82</td>
<td>100</td>
<td>2.11</td>
</tr>
<tr>
<td>1000</td>
<td>2000</td>
<td>100</td>
<td>9.69</td>
<td>100</td>
<td>5.26</td>
</tr>
<tr>
<td>2000</td>
<td>2000</td>
<td>100</td>
<td>14.04</td>
<td>100</td>
<td>12.73</td>
</tr>
<tr>
<td>4000</td>
<td>2000</td>
<td>100</td>
<td>23.71</td>
<td>100</td>
<td>32.58</td>
</tr>
<tr>
<td>8000</td>
<td>2000</td>
<td>100</td>
<td>48.34</td>
<td>100</td>
<td>79.75</td>
</tr>
<tr>
<td>16000</td>
<td>2000</td>
<td>100</td>
<td>210.03</td>
<td>100</td>
<td>234.53</td>
</tr>
</tbody>
</table>

Hamiltonian cycles (possibly only one). This is achieved through the use of an iterative method starting from a randomly generated Hamiltonian graph with one known Hamiltonian cycle. Then, by removing carefully selected edges iteratively, the graph remains Hamiltonian but some Hamiltonian cycles are eliminated. This process is continued until the algorithm is unable to find a Hamiltonian cycle other than the known Hamiltonian cycle. Algorithm 2 describes our method to modify randomly generated Hamiltonian graphs. We will demonstrate that some algorithms struggle to solve many of the resulting instances.

Since Algorithm 2 never removes any of the edges of the Hamiltonian Cycle $HC_i$, the graph remains Hamiltonian. Consequently, the algorithm will either stop once the chosen solver iterates $MaximumCount$ times without managing to find any Hamiltonian cycle other than the known one, or without finding any Hamiltonian cycle at all. Obviously, instances for which the known Hamiltonian cycle is found every time are not of interest and can be discarded, while instances where it often occurs that no Hamiltonian cycle is found are more difficult and may provide interesting benchmarking instances.

A key feature of the above algorithm is that the tours eliminated at each step are influenced by the algorithm being used. Therefore, if that algorithm is biased towards certain types of tours, those are likely to be among the first eliminated,
and the remaining tours may, in some sense, be more “difficult” for that algorithm to discover. In our testing, we found it was almost always the case that a modified random graph which was very difficult for the algorithm used to modify it, would be trivial to solve for all other tested algorithms. Nonetheless, Algorithm 2 provides a procedure for generating instances specifically constructed to stress the particular HCP or TSP algorithm being tested. Indeed, one measure of robustness for an algorithm is the degree to which it is impervious to this kind of algorithmic attack.

Algorithm 2: Producing modified random graphs

1. Set MaximumCount to some appropriately large number (in our implementation MaximumCount = 100). Set Count = 0.

2. Construct a random Hamiltonian graph $G$ with a known Hamiltonian Cycle $HC_i$ (there are numerous ways for constructing such a graph. A simple example is to start with a Hamiltonian cycle, and then randomly add edges to the graph).

3. Solve the graph $G$ using the chosen solver and find a Hamiltonian cycle $HC_r$. If no Hamiltonian cycle can be found, or if $HC_i = HC_r$, proceed to (3.1). Otherwise, go to step (4).

   3.1. Set Count = Count + 1;

   3.2. If Count > MaximumCount, then STOP and output $G$.

   3.3. Randomly relabel the vertices of $G$, keeping track of how this alters $HC_i$, and go back to the start of step (3).

4. Set Count = 0 and proceed to (4.1).

   4.1. Find an edge $e$ of $HC_r$ that is not contained in $HC_i$.

   4.2. Remove edge $e$ from $G$.

   4.3. Go back to step (3).

Benchmarking notes: The modified random graphs can be as large as desired.
They are usually very sparse and they contain many nodes of degree two. Concorde, which takes advantage of structure, seemed impetuous to this approach, and in fact performed better on average on the final graphs since they were sparse. SLH2 never reported any failures, however it often performed significantly slower as the algorithm progressed; this is explained by understanding that SLH2 works slower once the fast transformations are exhausted and opening transformations are required. LKH and CLK often reported failures, in some cases failing to find a tour in all final 100 iterations. Obviously, if the aim is to produce a benchmark set by this approach, the trivial instances should be discarded and only the difficult instances retained.

In Tables 3.8–3.11 below, we consider each algorithm individually. For various different graph sizes, we produced 2000 graphs using the above algorithm. For each of the 2000 graphs, we set MaximumCount = 100, and recorded the number of failures; that is, how often in the final 100 iterations that no Hamiltonian cycle was found. The following three ratios are reported in percentage:

- **Avg. Fail**: The average rate of failure for the 2000 samples.
- **Highest Fail**: The highest rate of failure reported for an instance among the 2000 samples.
- **Full success**: The percentage of samples where the solver succeeds in all 100 attempts.

The solution time is not reported for two reasons. First, constructing the modified random graphs is computationally expensive because often an algorithm is called thousands of times only to create a single instance. For practical reasons, we have used different CPUs to conduct the entire test in a reasonable time. As a result, the obtained execution times are not comparable with each other. Second, the reason for constructing these modified random graphs in the first place is to arrive at structures where an algorithm fails to be effective. Therefore, the focus
of Tables 3.8–3.11 is on whether an algorithm succeeds on these instances, rather than measuring the execution time.

**Table 3.8:** Performance of LKH on modified randomly generated HCP instances.

<table>
<thead>
<tr>
<th>Size</th>
<th>Sample</th>
<th>Avg. degree</th>
<th>Avg. Fail</th>
<th>Highest Fail</th>
<th>Full success</th>
</tr>
</thead>
<tbody>
<tr>
<td>250</td>
<td>2000</td>
<td>3.26</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>500</td>
<td>2000</td>
<td>3.30</td>
<td>0.71</td>
<td>43</td>
<td>76.2</td>
</tr>
<tr>
<td>1000</td>
<td>2000</td>
<td>3.32</td>
<td>11.85</td>
<td>88</td>
<td>44.51</td>
</tr>
<tr>
<td>2000</td>
<td>2000</td>
<td>3.36</td>
<td>35.60</td>
<td>97</td>
<td>35.40</td>
</tr>
<tr>
<td>4000</td>
<td>2000</td>
<td>3.33</td>
<td>59.07</td>
<td>100</td>
<td>28.27</td>
</tr>
</tbody>
</table>

**Table 3.9:** Performance of CLK on modified randomly generated HCP instances.

<table>
<thead>
<tr>
<th>Size</th>
<th>Sample</th>
<th>Avg. degree</th>
<th>Avg. Fail</th>
<th>Highest Fail</th>
<th>Full success</th>
</tr>
</thead>
<tbody>
<tr>
<td>250</td>
<td>2000</td>
<td>3.31</td>
<td>28.26</td>
<td>96</td>
<td>2.71</td>
</tr>
<tr>
<td>500</td>
<td>2000</td>
<td>3.31</td>
<td>65.45</td>
<td>99</td>
<td>0</td>
</tr>
<tr>
<td>1000</td>
<td>2000</td>
<td>3.34</td>
<td>93.54</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>2000</td>
<td>2000</td>
<td>3.35</td>
<td>99.96</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>4000</td>
<td>2000</td>
<td>3.39</td>
<td>100</td>
<td>100</td>
<td>0</td>
</tr>
</tbody>
</table>

**Table 3.10:** Performance of Concorde on modified randomly generated HCP instances.

<table>
<thead>
<tr>
<th>Size</th>
<th>Sample</th>
<th>Avg. degree</th>
<th>Avg. Fail</th>
<th>Highest Fail</th>
<th>Full success</th>
</tr>
</thead>
<tbody>
<tr>
<td>250</td>
<td>2000</td>
<td>3.29</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>500</td>
<td>2000</td>
<td>3.33</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>1000</td>
<td>2000</td>
<td>3.31</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>2000</td>
<td>2000</td>
<td>3.32</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>4000</td>
<td>2000</td>
<td>3.32</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
</tbody>
</table>

**Table 3.11:** Performance of SLH2 on modified randomly generated HCP instances.

<table>
<thead>
<tr>
<th>Size</th>
<th>Sample</th>
<th>Avg. degree</th>
<th>Avg. Fail</th>
<th>Highest Fail</th>
<th>Full success</th>
</tr>
</thead>
<tbody>
<tr>
<td>250</td>
<td>2000</td>
<td>3.33</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>500</td>
<td>2000</td>
<td>3.28</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>1000</td>
<td>2000</td>
<td>3.31</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>2000</td>
<td>2000</td>
<td>3.31</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>4000</td>
<td>2000</td>
<td>3.33</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
</tbody>
</table>
3.4 HCP Instances Corresponding to Other \(\mathcal{NP}\)-complete Problems

As mentioned earlier, HCP is \(\mathcal{NP}\)-complete, and is hence at least as difficult as any member of the vast set of NP problems. Hence, any instance of another NP problem can, in theory, be converted to an equivalent instance of HCP. It stands to reason that difficult instances of other problems, upon conversion, will result in difficult instances of HCP. It is common for such conversions to result in dramatic growth in size, even though that growth is bounded polynomially. However, in recent times there has been interest in looking specifically for conversions that only result in linear growth in size (e.g. see Creignou [32], Dewdney [34] and Filar et al [42]). Since our intention in this chapter is to not rely on problem size, all of the conversions we report on here result in only linear growth in the problem size. Although the growth is linear, in some cases there is a significant constant coefficient, so producing difficult instances of small size is non-trivial. Still, this approach often leads to difficult instances of moderate size.

3.4.1 Converting Computationally Difficult Problems to HCP

If an instance of an \(\mathcal{NP}\) problem is difficult, it is likely that the corresponding instance of HCP obtained by a conversion will be also difficult. Hence, we consider four such \(\mathcal{NP}\)-complete problems here, described below. The general idea of the conversions is described in [58]. Code to perform each of the conversions to HCP is available at [56].

**Chromatic Number Problem (COL):** The Chromatic Number problem requests the minimum number of colors required to color the vertices of a given graph, so that no edge in the graph has endpoints with the same color. The de-
cision variant of the problem asks whether such a vertex coloring is possible for a given number of colors $k$. For a fixed number of colors $k$, the conversion to HCP results in only linear growth in the size of the problem. In all tested examples, $k$ was set to 3.

**Generalized Instant Insanity (II):** Suppose you have $k$ cubes, where each face of each cube has a given color chosen from a set of $k$ colors. The Generalized Instant Insanity problem asks if it possible to stack these cubes in a column, oriented such that all $k$ colors can be seen along each of the four long faces. An instance with $k = 4$ was originally marketed as "Instant Insanity" by Parker Brothers, and the generalized version with arbitrary $k$ was subsequently shown to be NP-complete [105].

**n-Queens Problem (QN):** The $n$-Queens problem asks if, given an $n \times n$ chess-board, it is possible to place $n$ queens in such a way that none of the queens can take any of the other queens using their standard diagonal movements. The $n$-Queens problem is not NP-complete and it is known that the solution is possible for any integer $n \geq 4$. However, the problem is in NP and can hence be converted to HCP.

**Set splitting problem (SSP):** The Set splitting problem asks, for a given finite universe set $U$, and a family $F$ of subsets of $U$, if there exists a partition of $U$ into two disjoint non-empty subsets $V$ and $W$ such that each entry of $F$ contains at least one element from both $V$ and $W$.

For each of the above problems, we selected four instances to convert to into instances of HCP, ensuring we chose instances small enough that the converted graphs had fewer than 10,000 vertices. We then ran each of these instances 100 times for the four test heuristics. As can be seen in Table 3.12, CLK and LKH tended to find these instances difficult. Concorde and SLH2 were able to solve them up to a point, but slowed down significantly as the number of vertices in-
creased, with Concorde solving significantly faster than SLH2 in almost all tested cases.

**Table 3.12:** Performance of the four algorithms on converted computationally difficult problems to HCP. Solved columns report the number of times an optimal tour was obtained after 100 runs.

<table>
<thead>
<tr>
<th>Name</th>
<th>n</th>
<th>Concorde</th>
<th>CLK</th>
<th>LKH</th>
<th>SLH2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Solved</td>
<td>Time</td>
<td>Solved</td>
<td>Time</td>
</tr>
<tr>
<td>COL_1000</td>
<td>1000</td>
<td>100</td>
<td>5.49</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>COL_1950</td>
<td>1950</td>
<td>100</td>
<td>10.30</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>COL_4110</td>
<td>4110</td>
<td>100</td>
<td>42.81</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>COL_7998</td>
<td>7998</td>
<td>100</td>
<td>252.79</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>II_1002</td>
<td>1002</td>
<td>100</td>
<td>83.53</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>II_1992</td>
<td>1992</td>
<td>100</td>
<td>1573.68</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>II_3972</td>
<td>3972</td>
<td>0*</td>
<td>NA</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>II_7932</td>
<td>7932</td>
<td>0*</td>
<td>NA</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>QN_1044</td>
<td>1044</td>
<td>100</td>
<td>19.82</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>QN_1968</td>
<td>1968</td>
<td>100</td>
<td>136.78</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>QN_3894</td>
<td>3894</td>
<td>100</td>
<td>2039.75</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>QN_8544</td>
<td>8544</td>
<td>0*</td>
<td>NA</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>SSP_1011</td>
<td>1011</td>
<td>100</td>
<td>30.10</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>SSP_2007</td>
<td>2007</td>
<td>100</td>
<td>154.96</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>SSP_4050</td>
<td>4050</td>
<td>100</td>
<td>1393.69</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>SSP_8040</td>
<td>8040</td>
<td>100</td>
<td>12413.90</td>
<td>0</td>
<td>NA</td>
</tr>
</tbody>
</table>

### 3.5 Discussion

We now include a short analysis of the four tested algorithms, and their performance on the full benchmark set.

Concorde was very effective at finding optimal tours. It became less effective when the structure of the graph contained significant symmetries combined with low prevalence of optimal tours, but overall it was the most robust of the tested algorithms. Concorde is an exact algorithm, in the sense that it will not stop until an optimal tour is found. However, we have observed that sometimes the program halts on very difficult instances (for example, some generalized Petersen graphs), thereby possibly exposing some bugs in the implementation.
Chained Lin-Kernighan Algorithm is good at tackling random examples, but struggles when the underlying instances have a low number of Hamiltonian Cycles. It’s performance was dominated by that of LKH.

LKH is usually considered the best implementation of the Lin-Kernighan method [71]. It lived up to its reputation by outperforming Chained Lin-Kernighan Algorithm in all of our instances. It is often the quickest algorithm among the four algorithms and was usually able to output a solution quickly even when the instances were large. However, it struggled when the underlying instances had a low number of Hamiltonian Cycles. Combining this result with that of CLK seems to indicate that Lin-Kernighan type approaches struggle for these kinds of instances.

SLH2 was very reliable for finding Hamiltonian Cycles even in many of the structurally difficult graphs, with only the Fleischner graphs providing a consistent challenge. However, it was not as fast as Concorde or LKH, and was limited by the large amount of memory required for bigger instances.

Finally, the computational study of this chapter differentiates the relative performance of the different approaches for various instances. A natural question that arises is whether it is possible to combine these approaches effectively to complement their strengths and improve their weaknesses. We thoroughly discuss this question in the next chapter.
Chapter 4

The Sparse Traveling Salesman Problem

4.1 Introduction

Consider a weighted graph $G = (V, E, \omega)$ where $V$ is the set of nodes, $E$ is the set of edges between these nodes, and $\omega : E \rightarrow \mathbb{R}$ is a function that maps the edges of the graph to their weights. We now recall a few terms from Chapter 2. If $G$ is not a complete graph, there exists $v, x \in V$ such that $(v, x) \notin E$. Such an edge is referred as a penalty edge. A tour $T$ of $G$ is a set of edges that form a simple cycle of size $n$ where for every $(u, v) \in T$, nodes $u$ and $v$ are nodes of $G$ (i.e, $u, v \in V$). If a tour contains a penalty edge $(v, x)$, then $(v, x)$ is referred as a gap in that tour. The number of gaps in a tour $T$ is denoted by $g(T)$. The length of a tour in a sparse graph $G$ is defined to be

$$|T|_G := \sum_{(u,v)\in(T\cap E)} \omega(u,v)$$

(4.1)

Recall Definition 1.4.2 of the Sparse Traveling Salesman Problem (STSP).
Given a weighted graph \( G = (V, E, \omega) \), suppose \( g_{\text{min}} \) is the minimum number of gaps for any tour of \( G \). The STSP is the problem of finding the shortest tour with \( g_{\text{min}} \) gaps.

Note that if \( g_{\text{min}} = 0 \), the STSP is identical to the classical formulation of the TSP on a Hamiltonian graph. The STSP can be seen as an extension of the problem to graphs that may not contain Hamiltonian Cycles. To be able to compare tours with different number of gaps, we say that a tour \( T_1 \) is shorter than \( T_2 \) if \( g(T_1) < g(T_2) \).

The STSP may be easily converted to TSP. Let graph \( G' = (V, E', \omega') \) be an instance of TSP constructed as follows. Suppose \( P \) is a strict upper bound to the value of any tour, i.e., \( P > \text{len}(T) \) for any tour \( T \). For example, let

\[
P = 1 + \sum_{u \in V} \max_{(u,v) \in E} (\omega(v, u)).
\]

We will refer to \( P \) as the penalty value. Consider the set of weighted edges \( E' = \{(v, w)\mid v, w \in V\} \), which is the set of edges for a complete graph defined by nodes of \( V \). The weight of an edge \((v, x)\) in \( E' \) is:

\[
\omega'(v, x) = \begin{cases} 
\omega(v, x) & \text{if } (v, x) \in E, \\
P & \text{otherwise.}
\end{cases}
\]

For any tour \( T \) denote by \( |T|_{G'} \) the length of the tour \( T \) in \( G' \). The instance of TSP on \( G' \) is equivalent to the instance of STSP on \( G \) in the following sense.

**Proposition 4.1.1.** Tour \( T^{\text{opt}} \) is an optimal tour for the STSP instance on \( G = (V, E, \omega) \), if and only if, \( T^{\text{opt}} \) is also an optimal tour for the TSP instance on \( G' = (V, E', \omega') \).
Proof. Let $T$ be any tour of $G$ and $G'$. We have $|T|_{G'} = g(T)P + |T|_G$. Let $T^{opt}$ be an optimal solution to the instance of STSP on $G = (V, E, \omega)$. Then $|T^{opt}|_G \leq |T|_G$. Therefore for any tour $T$,

$$|T^{opt}|_{G'} = g_{\text{min}}P + |T^{opt}|_G \leq g(T)P + |T|_G = |T|_{G'}.$$  

Therefore if $T^{opt}$ is an optimal solution to the instance of STSP on $G = (V, E, \omega)$, then $T^{opt}$ is an optimal solution to the instance of TSP on $G = (V, E', \omega')$ as well.

Now consider the opposite where $T^{opt}$ is an optimal solution to the instance of TSP on $G' = (V, E', \omega')$. Let $T$ be a tour such that $g(T) = g_{\text{min}}$. By optimality of $T^{opt}$ in $G'$,

$$|T^{opt}|_{G'} \leq |T|_{G'} = g_{\text{min}}P + |T|_G < (g_{\text{min}} + 1)P.$$  

In order for the above inequality to hold, we must have $g(T^{opt}) = g_{\text{min}}$. Then

$$|T^{opt}|_G = |T^{opt}|_{G'} - g_{\text{min}}P \leq |T|_{G'} - g_{\text{min}}P = |T|_G.$$  

Therefore if $T^{opt}$ is an optimal solution to the instance of TSP on $G$, then $T^{opt}$ is an optimal solution to the instance of STSP on $G$. □

In this chapter we discuss algorithmic approaches to solve STSP, and some applications of STSP. Because the STSP formulation is so easily convertible to a TSP formulation by merely adding penalty edges, the literature on TSP and STSP is largely undifferentiated. Many TSP algorithms, such as Concorde, allow the user to input the graph data in sparse format and then penalty edges are added automatically. However, a closer look at solving sparse instances of the TSP highlights particular difficulties of these instances. Most notably, the difficulties that arise from structure, rather than size, are often more significant in STSP compared to the general form of TSP. For example, for an instance of TSP on a complete graph, any permutation of nodes constitutes a feasible solution. This gives heuristic algorithms a great degree of flexibility, as any heuristic algorithm should in principle
be able to freely search through many feasible solutions. In contrast, for an instance of TSP on a sparse graph, even obtaining a single tour with the minimal number of gaps (even if it is a sub-optimal tour) is $\mathcal{NP}$-Hard. As demonstrated in Chapter 3, these structural difficulties in some cases pose significant challenges to TSP solvers.

The remainder of this chapter is laid out as follows. I discuss the existing approaches that are relevant to solving sparse instances of TSP in Section 4.2. I also examine ways of complementing these approaches with new methods in order to overcome the structural difficulties of STSP instances. Furthermore, there is a broad range of applications for algorithmic solutions to STSP. The first example, discussed in Section 4.3, is the conversion of a well-known variation TSP, called the Time Dependent Traveling Salesman Problem, to STSP. The second example, discussed in Section 4.4, is an STSP formulation for the DNA assembly problem.

### 4.2 A Hybrid Approach to Solving TSP

In the past few decades, researchers have generated a vast body of literature on a diverse range of algorithms for solving the TSP. Of course, these studies are also highly relevant to the study of STSP. The objective of this section is to propose an effective combination of existing and new methods - a hybrid approach - for solving STSP.

Many of the state-of-the-art TSP algorithms consist of multiple algorithmic approaches. For example, Concorde [7] uses various methods including (but not limited to) linear programming, branch and bound, and an implementation of the Lin-Kernighan algorithm [9]. Also, LKH [62], which is one of the best implementations of the Lin-Kernighan algorithm, uses preprocessing algorithms, such
as sparsification algorithms and sub-gradient optimization, to improve the tour-improvement choices during the Lin-Kernighan stage. We have noticed that without the extra modules, the performance of LKH is significantly affected.

A close look into the algorithms that solve TSP reveals that sparsification has always been a critical part of the algorithmic solutions to TSP. For instance, practically all variations of Lin-Kernighan sparsify an instance before applying their tour-improvement techniques. The original Lin-Kernighan heuristic [88] only considered the 5 nearest neighbors to limit the search space. As a more sophisticated example, one of the primary contributions of LKH [62] was to introduce a novel measure called \( \alpha \)-nearness that is used to sparsify the graph prior to starting the Lin-Kernighan algorithm. Concorde also constructs its linear programs (LPs) using a subgraph of the original problem that only contains a subset of the edges contained in the graph. These sparsifications are useful, and often necessary, from both time efficiency and memory management prospective.

As discussed in Chapter 3, the performance of TSP heuristics has been extensively studied. To identify the components of an effective approach for solving the STSP, we have reviewed the data on the performance of various algorithms solving instances of TSP [71, 125, 104, 57]. We have also considered various algorithmic approaches such as the Lin-Kernighan heuristic [62], genetic algorithms [96], SLH [10] and the collection of approaches in Concorde [7]. We have identified four distinct modules which can contribute to an effective hybrid approach for solving STSP.

**Helper algorithm**

1. **Dynamic sparsification**: Involves a process that creates a sparsified graph. The process continues to run throughout the rest of the algorithm, and the sparsified graph will be updated if the process finds an alternative sparsified graph which is better by some well-defined measure.
Upper bound algorithm

2. **Tour Improvement Transformations**: Involves a process that uses transformations which transform the current tour to another tour with an improved value.

3. **BIS-based transformations**: A process that minimizes the number of edges that do not belong to the optimal tour on the current tour. For example, the process removes penalty edges on the current tour which do not belong to the sparse graph.

Lower bound algorithm

4. **Optimal tour verification**: A process that verifies the optimality or measures the quality of a tour by obtaining and iteratively improving lower bound values.

We now discuss each of these four modules in detail.

### 4.2.1 Dynamic sparsification module

Consider a relaxed LP formulation of TSP where each edge is represented by a continuous variable. For example, Concorde uses the following LP to initiate its process of finding a lower bound.

\[
\min \sum_{i=1}^{n} \sum_{j=1, j \neq i}^{n} c_{ij}x_{ij} : \text{minimize the length of the tour}
\]

subject to:

\[
\begin{cases}
\sum_{j \in V} x_{ij} = 2, & \forall i \in V, \\
0 \leq x_{ij} \leq 1 & \forall i, j \in V.
\end{cases}
\]
The solution to an LP relaxation of TSP can be calculated efficiently. Accordingly, let us call the subgraph containing only those edges which correspond to the non-zero variables in the LP solution the support graph induced by the LP solution. Edges corresponding to variables with an objective coefficient of zero do not exist in the support graph, and in this way the support graph is a sparsified subgraph of the underlying graph. If an optimal solution of the relaxed formulation corresponds to a tour in the graph, the support graph is the optimal tour. In this case, the sparsification is an ideal sparsification.

Where the LP solution provides a near-optimal lower bound for the TSP, we expect that many edges of the optimal tour should overlap with edges of the support graph. Since for “tight enough” constraints, the optimal tour will be fully contained in the support graph, we expect an effective process of tightening the constraints of an LP, will lead to an increase in the number of overlaps. The better the constraints are, the more sparsified the support graph is, and the more edge overlaps there are between the support graph and the optimal tour. For this reason, constructing a support graph seems to be an promising method of graph sparsification. When a good support graph is found, we may use it to assist heuristics that find the upper bound.

The quality of the sparsification depends on the quality of the LP relaxation. We generally expect that if an optimal solution of the LP formulation provides a better lower bound, the corresponding sparsification is of higher quality. This gives us a clear measure for comparing the quality of two support graphs. Therefore, to obtain better sparsifications by constructing support graphs, we need a process which dynamically improves the LP relaxations of TSP.

Exact TSP algorithms such as Concorde construct a polytope that approximates the TSP polytope, and seek to iteratively improve it. Techniques for approximating the TSP polytope with an LP model, and improving it iteratively, are
extensively studied. We refer the reader to [7] for an excellent study on this topic. These polytopes have a corresponding support graph which may be used for dynamically sparsifying the TSP graph by simply updating the support graph once there is a large enough improvement in the polytope as measured by improvement in the lower bound.

We note that Concorde’s implementation is aimed at finding the exact solutions of TSP. This objective is in line with our aim of obtaining the best sparse graph possible. However, finding a good support graph is a far easier objective as it only relies on relaxed solutions. Hence, in the context of solving large graphs, it is possible to avoid some of the computationally expensive elements of Concorde while still obtaining improved lower bounds and consequently support graphs. Obtaining an exact solution is incredibly difficult for large graphs, but dynamic sparsification may still be practical to assist the upper bound algorithms in limiting their search space. Investigating the strategies for TSP relaxations that affect the quality and efficiency of the dynamic sparsification is a topic for future exploration.

4.2.2 Tour value improvement module

After limiting the search space through sparsification, we need to obtain an upper bound for the TSP. Consider a tour $T$ of the graph $G$ with $n$ nodes. A tour improvement algorithm tries to find a set of edges $X = \{x_1 = (a_1, a_2), \ x_2 = (a_3, a_4), \ldots\}$ of the tour $T$ and exchanges them with an alternative set of edges $Y = \{y_1 = (b_1, b_2), \ y_2 = (b_3, b_4), \ldots\}$ not contained in $T$, so that the exchange results in a new tour $T'$ with a lower value than $T$. Of course, this exchange must lead to another feasible tour to be considered a feasible exchange.

Suppose that an algorithm constructs the sets $X$ and $Y$ iteratively by adding an edge $x_i$ to $X$ and edge $y_i$ to $Y$ edge at iteration $i$. If for all $i$, $x_i$ and $y_i$ share an endpoint, and if $x_{i+1}$ exist then $x_{i+1}$ and $y_i$ share an endpoint, then the exchange
of \((T \setminus X) \cup Y\) is known as a \textit{sequential exchange}, given the exchange is feasible. Sequential exchanges are the central technique that is used in the Lin-Kernighan algorithm because they allow fast exchanges with multiple edges. Though this kind of exchange is very effective in finding local minimums, sometimes non-sequential exchanges are required to find better local minimums, or the global minimum. Figures 4.1–4.2 illustrate examples of a sequential exchange and a non-sequential exchange respectively.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{sequential_exchange.png}
\caption{Example of a sequential transformation.}
\end{figure}

In the Lin-Kernighan algorithm, the set \(X\) is constructed iteratively and in each step there are only up to \(d\) choices for the next edge, where \(d\) is the maximum degree of a node in the graph. For a sparse graph searching for a sequential \(k\)-exchange from a gap is at worst \(O(d^k)\). This space may further be reduced substantially by disabling backtracking, and enforcing conditions that filter certain choices (e.g., the gain criterion in the Lin-Kernighan algorithm) to the point that the search space is reduced to only \(O(dk)\). The important advantage of sequential exchanges over non-sequential exchanges is that for sparse graphs, looking for a sequential \(k\)-exchange depends on \(k\) and \(d\), rather than the size of the graph \(n\). This enables a powerful and computationally practical tool for tour-improvement even when \(k\) is large.

There are situations where sequential exchanges are not sufficient for solving the graph. Non-sequential improvements remain difficult to find in an efficient way.
and are often hard coded in a limited form in the implementations of the Lin-Kernighan algorithm. Since the space of non-sequential exchanges grow exponentially, the Lin-Kernighan type algorithms often encounter difficulties in instances where these transformations are required. Indeed, the benchmarking exercise in Chapter 3 illustrates how small yet structurally difficult instances of HCP may provide significant challenges for even the best Lin-Kernighan implementations.

Nevertheless, the success of Lin-Kernighan type algorithms is remarkable. The most successful implementation of Lin-Kernighan heuristic is currently Helsgaun’s implementation of the Lin-Kernighan heuristic, entitled LKH [62]. Helsgaun’s implementation has proven to be able to obtain solutions within 1% of the optimal tour, for problems of up to 10 million cities. It also currently holds the record for best known solutions for examples of up to 10 million nodes examples with unknown optima in the DIMACS challenge. For the tour improvement component of our method, we have used the LKH code provided on the LKH web-page [61].

4.2.3 Gap Reduction Module

One of the difficulties of finding the optimal tour in a sparse graph is finding any tour with minimal gaps. If the graph is Hamiltonian, even finding one Hamiltonian
Cycle is an \( \mathcal{NP} \)-complete problem. In the general case, finding one tour with the minimal number of gaps is \( \mathcal{NP} \)-Hard.

The sequential exchanges invented by Lin and Kernighan [88] are the primary method in tour improvement algorithms for solving the TSP. An important question is, how effective is this technique in solving sparse instances of TSP? Also, could tour improvement algorithms benefit from using BIS-based transformations as well? We refer the reader to Chapter 2 for the main discussion of the BIS-based transformations.

Table 4.1 reports the performance of LKH tested on 1000 instances for five different sizes from 250 to 4000 nodes. These instances are randomly generated graphs where every node is adjacent to exactly three nodes. The weights are randomly assigned positive integers of up to 100. Using the default settings, which involves 10 randomized runs and pre-solving with sub-gradient optimization, LKH demonstrates an impressive capability in finding Hamiltonian Cycles. In the following tables, columns \( HC \) and \( Opt. \) represent the percentage of runs where a Hamiltonian cycle and an optimal tour are obtained respectively. The \( Time \) column reports the execution time from calling the program until the conclusion of the program.

**Table 4.1**: Average performance of LKH on 1000 random regular TSP graphs with degree 3. Columns \( HC \) and \( Opt. \) represent the percentage of runs where a Hamiltonian cycle and an optimal tour are obtained respectively. The \( Time \) column reports the average execution time of the program for a single run.

<table>
<thead>
<tr>
<th>Size</th>
<th>LKH</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>HC</td>
<td>Opt.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>250</td>
<td>100%</td>
<td>100%</td>
<td>0.60</td>
<td></td>
</tr>
<tr>
<td>500</td>
<td>100%</td>
<td>100%</td>
<td>2.70</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>99.9%</td>
<td>99.7%</td>
<td>14.07</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>99%</td>
<td>97.5%</td>
<td>81.10</td>
<td></td>
</tr>
<tr>
<td>4000</td>
<td>100%</td>
<td>35%</td>
<td>592.45</td>
<td></td>
</tr>
</tbody>
</table>

It is important to emphasize that this strong performance is the result of the
effective integration of several techniques, not just the Lin-Kernighan tour improvement transformations. A fair assessment of the Lin-Kernighan transformations should remove the impact of other algorithms that assist the Lin-Kernighan transformations in obtaining higher quality tours. To this end, we conducted the following experiment. We ran LKH again on the same instances as those in Table 4.1, but with one of its important techniques turned off. Specifically, we ran LKH with two alternative sets of parameters. For parameter settings 1, sub-gradient optimisation is disabled and only one run is allowed (randomized runs are disabled). For parameter settings 2, sub-gradient optimisation is disabled but the default 10 randomized runs are allowed.

It can be seen in Table 4.2 that the performance significantly suffers when sub-gradient optimisation is not used, particularly as the size of the instance increases. Even though parameter setting 2 performs slightly better than the performance of parameter setting 1, a comparison with Table 4.1 shows that using the randomized runs does not compensate for removing the sub-gradient optimisation. These findings reveal that the performance of Lin-Kernighan transformations on sparse instances relies heavily on their integration with other techniques such as sub-gradient optimisation.

We also ran SLH on the instances of Table 4.2. SLH performs extremely well, both in terms of efficiency, and success rate at obtaining Hamiltonian cycles. Of course, SLH is not designed to obtain an optimal tour, but it may still be valuable if merely a tour with minimal gaps is desired. By comparison, for the large graphs, even discovering a tour with minimal gaps proved extremely difficult for LKH without the assistance of sub-gradient optimisation.

The results of the two tables above point to two important conclusions. First, the comparison of LKH results in Table 4.1 and Table 4.2 demonstrates that tour improvement techniques like the sequential Lin-Kernighan transformations can be
Table 4.2: Comparative performance of SLH and two altered settings of LKH on random regular graphs. For LKH parameter setting 1 and 2, subgradient optimization is disabled. For the former, 1 run, and for the latter, 10 randomized are allowed. Columns HC and Opt. represent the percentage of runs where a Hamiltonian cycle and an optimal tour are obtained respectively. The Time column reports the average execution time of the program for a single run.

<table>
<thead>
<tr>
<th>Size</th>
<th>LKH setting 1</th>
<th></th>
<th>LKH setting 2</th>
<th></th>
<th>SLH2</th>
</tr>
</thead>
<tbody>
<tr>
<td>250</td>
<td>74.0%</td>
<td>5.5%</td>
<td>0.08</td>
<td>74.0%</td>
<td>5.6%</td>
</tr>
<tr>
<td>500</td>
<td>67.8%</td>
<td>0.7%</td>
<td>0.37</td>
<td>67.9%</td>
<td>0.8%</td>
</tr>
<tr>
<td>1000</td>
<td>50.8%</td>
<td>0.1%</td>
<td>2.17</td>
<td>50.8%</td>
<td>0.1%</td>
</tr>
<tr>
<td>2000</td>
<td>29.2%</td>
<td>0.0%</td>
<td>13.07</td>
<td>29.2%</td>
<td>0.01%</td>
</tr>
<tr>
<td>4000</td>
<td>9%</td>
<td>0%</td>
<td>82.21</td>
<td>9%</td>
<td>0%</td>
</tr>
</tbody>
</table>

enhanced by combining them with other strategies. The relative dominance of SLH in Table 4.2 also suggest that there are situations where BIS-based transformations provide some advantage over Lin-Kernighan transformations. For these two reasons, combining the two techniques seem to be a promising direction for enhancing tour improvement algorithms.

4.2.4 Verification Module

Any valid tour provides an upper bound for the length of the optimal tour. If a lower bound value can be found which is equal to the upper bound provided by a tour, then that tour must be optimal. Even though various heuristic methods may be used to obtain a good solution, it is the verification process captures the computational difficulty of TSP. A lower bound is often obtained by solving LP relaxations of the IP formulations of TSP.

If the cleverly designed LP relaxations of TSP do not output a tour, verification typically proceeds via a computationally expensive branch and bound process. The branch and bound process is the standard method for solving ILP and it is guaranteed to find the optimal solution in exponential time. For large instances of TSP, this process is often intractable, and so we may forgo the full verification
of the tour to obtain a near-optimal solution. In this case, the lower-bound solution that is not equal to the upper-bound solution is still valuable, as the gap between the lower bound and upper bound is the standard measure of the quality of a heuristic solution. Note that if the verification is not concluded due to computational difficulty, the hybrid method acts as a heuristic method.

4.2.5 Implementation

In light of the above discussions, we have produced several software implementations designed for STSP inputs. All of the following programs may be accessed at https://www.travelingsalesman.org.

- \textit{sLKH}

The standard version of LKH does not allow inputs of general sparse graphs. I have augmented the open source code of LKH\footnote{The LKH code is available for academic and non-commercial use at http://akira.ruc.dk/~keld/research/LKH/. The author of LKH reserves all rights to the code.} to accommodate STSP with any weights. It is possible to input any sparse graph as a complete graph, however this requires large amounts of memory. For this reason, I have incorporated AVL-trees to store the weights of the graph which reduce the memory requirement from $O(n^2)$ to $O(nd)$, where $n$ is the size of the graph and $d$ is the maximal degree of the graph. For example, storing a 10,000 node graph using a full matrix could require up to 6.4 GB of RAM while AVL-tree could save up to 99\% of the memory usage, even if the degree of the graph is as large as 100. On the other hand, the worst-case time taken by query for weights is altered from $O(1)$ to $O(log(n))$. This trade-off ends up being insignificant because LKH uses candidate sets which ensure that most queries are $O(1)$. In addition to the use of AVL-trees, minor changes to the LKH code have been made to ensure maximum efficiency for STSP. The sLKH software operates both on both Windows and Linux.
• **sLKH-SLH**

This code is a combination of sLKH and SLH2 (described in Chapter 2) that is implemented for both Windows and Linux. For difficult instances where finding a tour in the sparse graph is difficult, or for instances where sLKH is unable to reduce the number of gaps to their minimal, the SLH algorithm is activated. After SLH concludes its process, LKH is re-activated on the best tour obtained by SLH to find a local minimum.

• **Concorde-SLH Hybrid**

The Concorde-SLH hybrid algorithm is the proof of concept version of the algorithm that incorporates all of the 4 modules described in this section, and it is implemented for Linux. Two parallel runs are dynamically integrated to find the lower bound and upper bound for the graph as described in this section. For the lower bound we have used the implementation of Concorde 03.12.19\(^2\). The support graph is then created from the solution of the LP problem and passed on to SLH to solve. One unit of improvement on the lower bound is set to be the trigger for updating the support graph. The upper bound value and the best tour is either updated by SLH or by in-build features of Concorde. When an upper bound matches a lower bound, the tour is returned as the optimal tour.

• **sTSP platform**

A unified platform is implemented which unifies the input and output format of sparse instances for multiple sparse TSP algorithms. The solver may be chosen from sLKH, sLKH-SLH, Concorde-SLH Hybrid or Concorde algorithm. Furthermore, independent randomized test may be paralleled under this platform.

\(^2\)Concorde source code is available for academic research use program at http://www.math.uwaterloo.ca/tsp/concorde/.
4.2.6 Computational Study on Instances of Chapter 3

In Table 4.3, we report the performance of Concorde-SLH Hybrid and sLKH-SLH on the benchmarking instances described in Chapter 3 ran on the same computer with the same constraints as in Chapter 3.

A comparison with Tables 3.3 - 3.12 of Chapter 3 reveals a substantial improvement as a result of the hybrid methods. On virtually all of these instances, the Concorde-SLH and sLKH-SLH outperform all of the algorithms tested in Chapter 3. The results are most striking on examples such as GP3_1002, where Concorde, LKH, and CLH failed to output a solution. SLH obtained a solution after 1284.23 seconds. In contrast, for solving GP3_1002, there is a 86% and 99% reduction on the the computational time required by Concorde-SLH and sLKH-SLH respectively compared to SLH. Overall, sLKH-SLH reflects the strengths of both SLH and sLKH; a quick solution is obtained by sLKH when possible, and SLH overcomes some of the trickier instances. Concorde-SLH is the most robust method and provided valid solutions to the largest subset of the problems. The only instances were Concorde-SLH failed to output a solution were some instances of the notoriously difficult Fleischner graphs. Even for these instances, Concorde-SLH found valid solutions to FLS_170 for which all other algorithms failed to obtain a solution.

4.3 Application 1: Time-Dependant Traveling Salesman Problem

In this section, we investigate the use of an STSP formulation for solving the Time-Dependant Traveling Salesman Problem (TDTSP). TDTSP is an extension of the TSP where the distances between pairs of cities are not fixed. Rather, the distance between a pair of cities depends upon the position in which the corresponding edge
Table 4.3: Performance of Concorde-SLH and sLKH-SLH on Chapter 3 instances. Solved columns report the number of times an optimal tour was obtained after of 100 runs.

<table>
<thead>
<tr>
<th>Name</th>
<th>n</th>
<th>Concorde-SLH</th>
<th>sLKH-SLH</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Solved</td>
<td>Time (s)</td>
</tr>
<tr>
<td>GPN_122</td>
<td>122</td>
<td>100</td>
<td>0.06</td>
</tr>
<tr>
<td>GPN_244</td>
<td>244</td>
<td>100</td>
<td>0.05</td>
</tr>
<tr>
<td>GPN_482</td>
<td>482</td>
<td>100</td>
<td>0.05</td>
</tr>
<tr>
<td>GPN_998</td>
<td>998</td>
<td>100</td>
<td>0.14</td>
</tr>
<tr>
<td>GP3_126</td>
<td>126</td>
<td>100</td>
<td>0.07</td>
</tr>
<tr>
<td>GP3_246</td>
<td>246</td>
<td>100</td>
<td>0.05</td>
</tr>
<tr>
<td>GP3_486</td>
<td>486</td>
<td>100</td>
<td>0.05</td>
</tr>
<tr>
<td>GP3_1002</td>
<td>1002</td>
<td>100</td>
<td>0.05</td>
</tr>
<tr>
<td>GP0_130</td>
<td>130</td>
<td>100</td>
<td>0.38</td>
</tr>
<tr>
<td>GP0_250</td>
<td>250</td>
<td>100</td>
<td>87.96</td>
</tr>
<tr>
<td>GP0_490</td>
<td>490</td>
<td>100</td>
<td>1.96</td>
</tr>
<tr>
<td>GP0_1006</td>
<td>1006</td>
<td>100</td>
<td>787.28</td>
</tr>
<tr>
<td>SH_64</td>
<td>64</td>
<td>100</td>
<td>1.13</td>
</tr>
<tr>
<td>SH_125</td>
<td>125</td>
<td>100</td>
<td>0.06</td>
</tr>
<tr>
<td>SH_250</td>
<td>250</td>
<td>100</td>
<td>0.08</td>
</tr>
<tr>
<td>SH_500</td>
<td>500</td>
<td>100</td>
<td>0.122</td>
</tr>
<tr>
<td>SH_1000</td>
<td>1000</td>
<td>100</td>
<td>0.05</td>
</tr>
<tr>
<td>SN_124</td>
<td>124</td>
<td>100</td>
<td>2.34</td>
</tr>
<tr>
<td>SN_252</td>
<td>252</td>
<td>100</td>
<td>0.61</td>
</tr>
<tr>
<td>SN_500</td>
<td>500</td>
<td>100</td>
<td>9.88</td>
</tr>
<tr>
<td>SN_1004</td>
<td>1004</td>
<td>100</td>
<td>5.40</td>
</tr>
<tr>
<td>FLS_170</td>
<td>170</td>
<td>100</td>
<td>7574</td>
</tr>
<tr>
<td>FLS_338</td>
<td>338</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>FLS_507</td>
<td>507</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>FLS_676</td>
<td>676</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>FLS_845</td>
<td>845</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>FLS_1014</td>
<td>1014</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>FLS3_408</td>
<td>408</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>FLS3_731</td>
<td>731</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>FLS3_1054</td>
<td>1054</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>Name</td>
<td>n</td>
<td>Concorde-SLH</td>
<td>sLKH-SLH</td>
</tr>
<tr>
<td>---------</td>
<td>-----</td>
<td>--------------</td>
<td>----------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Solved</td>
<td>Time (s)</td>
</tr>
<tr>
<td>COL_1000</td>
<td>1000</td>
<td>100</td>
<td>2.31</td>
</tr>
<tr>
<td>COL_1950</td>
<td>1950</td>
<td>100</td>
<td>4.34</td>
</tr>
<tr>
<td>COL_4110</td>
<td>4110</td>
<td>100</td>
<td>11.57</td>
</tr>
<tr>
<td>COL_7998</td>
<td>7998</td>
<td>100</td>
<td>43.87</td>
</tr>
<tr>
<td>II_1002</td>
<td>1002</td>
<td>100</td>
<td>11.15</td>
</tr>
<tr>
<td>II_1992</td>
<td>1992</td>
<td>100</td>
<td>278.07</td>
</tr>
<tr>
<td>II_3972</td>
<td>3972</td>
<td>100</td>
<td>2075.04</td>
</tr>
<tr>
<td>II_7932</td>
<td>7932</td>
<td>0</td>
<td>NA</td>
</tr>
<tr>
<td>QN_1044</td>
<td>1044</td>
<td>100</td>
<td>5.42</td>
</tr>
<tr>
<td>QN_1968</td>
<td>1968</td>
<td>100</td>
<td>60.75</td>
</tr>
<tr>
<td>QN_3894</td>
<td>3894</td>
<td>100</td>
<td>318.50</td>
</tr>
<tr>
<td>QN_8544</td>
<td>8544</td>
<td>100</td>
<td>5836.16</td>
</tr>
<tr>
<td>SSP_1011</td>
<td>1011</td>
<td>100</td>
<td>9.40</td>
</tr>
<tr>
<td>SSP_2007</td>
<td>2007</td>
<td>100</td>
<td>47.03</td>
</tr>
<tr>
<td>SSP_4050</td>
<td>4050</td>
<td>100</td>
<td>261.93</td>
</tr>
<tr>
<td>SSP_8040</td>
<td>8040</td>
<td>100</td>
<td>1964</td>
</tr>
</tbody>
</table>

appears on the tour. Multiple formulations of the TDTSP have been studied in the literature [101, 44, 49] and various algorithms have been developed to solve the problem [30, 1].

The problem can be thought of as follows. Consider $n$ cities labeled 1 to $n$. Suppose a traveling salesman needs to start from one of these cities, visit all of them exactly once, and then returns to the starting city. Also, the distance of travel from city $i$ to city $j$ is given by $d(i, j, t)$, where $t$ represents the step at which the travel from city $i$ to $j$ takes place. For the travel between the first pair of cities $t = 1$, for the travel between the second pair of cities $t = 2$, and so on, until the salesman returns to the initial city at step $t = n$. The distances between cities at all steps are known in advance and may be stored in a 3-dimensional matrix. Then, the TDTSP may be formally defined as follows.

**Definition 4.3.1. Time Dependent Traveling Salesman Problem (TDTSP)**

Given $n$ cities and the step-dependant distances between them, the TDTSP is the problem of finding the shortest route that starts from one of the cities, visits each
city exactly once, and then returns to the initial city.

A simple ILP formulation of TDTSP is as follows (paraphrased from [66]). Let

\[
x_{ij}^t = \begin{cases} 
1 & \text{if at step } t \text{ the salesman travels from city } i \text{ to city } j, \\
0 & \text{otherwise.}
\end{cases}
\]

Then the objective function is

\[
\min \sum_{t=1}^n \sum_{i=1}^n \sum_{j=1, j \neq i}^n x_{ij}^t d(i, j, t),
\]

subject to

\[
\begin{align*}
\sum_{i=1, i \neq j}^n \sum_{t=1}^n x_{ij}^t &= 1, & \text{for } j = 1 \ldots n. \\
\sum_{i=1, i \neq j}^n x_{ij}^t &= \sum_{k=1, k \neq j}^n x_{jk}^{t+1}, & \text{for } j = 1 \ldots n, \ t = 1 \ldots n - 1, \\
\sum_{i=1, i \neq j}^n x_{ij}^n &= \sum_{k=1, k \neq j}^n x_{jk}^1, & \text{for } j = 1 \ldots n. \\
x_{ij}^t &\geq 0 \text{ and integer} & \text{for } i = 1 \ldots n, \ j = 1 \ldots n, \ t = 1 \ldots n.
\end{align*}
\]

To represent an instance of TDTSP by a graph, we construct the weighted directed graph \( \Gamma = (V, E, \omega) \) where \( V \) is the set of nodes, \( E \) is the set of directed edges, and \( \omega : E \to \mathbb{R} \) is a function that maps the edges to their weights. A node \( v \) is a pair \( v \equiv [i, t] \) representing city \( i \) at step \( t \). Then the directed edge \( ([i, t], [j, t]) \) represents the travel from city \( i \) to city \( j \) at step \( t \) and its weight is equal to \( d(i, j, t) \). If the salesman chooses to travel on the \( ([i, t], [j, t]) \) edge, he leaves city \( i \) and enters city \( j \) at step \( t \). Let the cluster of nodes \( C_i = \{ v \mid v \equiv [i, t], \ t = 1, \ldots n \} \), be the set of all nodes associated with the city \( i \) for different steps. Then the set of nodes and edges of \( \Gamma \) are as follows:

\[
V = \{ v \mid v \equiv [i, t], \ \text{for } i = 1, 2, \ldots n, \ t = 1 \ldots n \},
\]
\[ E = \{(u, v) \mid u \equiv [i, t_1], v \equiv [j, t_2], \text{ such that } u, v \in V \land t_1 = t_2\}, \]
\[ \omega(u, v) = d(i, j, t) \quad \forall (u \equiv [i, t], v \equiv [j, t]) \in E. \]

Graph \( \Gamma \) has \( n \) connected components where each component is associated with a step and represents the movement from pairs of nodes at the corresponding step. An instance of TDTSP is then unambiguously defined by \( \Gamma \).

Next, we define a \textit{TDTSP tour} \( \Upsilon \) be be any set of edges that correspond to the route of the traveling salesman satisfying the conditions for a valid route of the TDTSP. These conditions may be described as follows:

\textbf{Condition 1:} The salesman enters and leaves each city exactly once. That is, for any city \( i \), there must be exactly two distinct cities \( h \) and \( j \) where 
\((\{h, t_1\}, \{i, t_1\}) \in \Upsilon \) and 
\((\{i, t_2\}, \{j, t_2\}) \in \Upsilon \).

\textbf{Condition 2:} If the salesman enters a city at step \( t \), the salesman leaves the city at step \( 1 + t \mod n \). For some \( h \), if \((\{h, t\}, \{i, t\}) \in \Upsilon \) then for some \( j \),
\((\{i, (1 + t \mod n)\}, \{j, (1 + t \mod n)\}) \in \Upsilon \).

\textbf{Condition 3:} The traveling salesman must visit all cities consecutively (each after another until all cities are visited).

Condition 3 is necessary because the \textit{TDTSP tour} is defined as a set rather than a sequence, and ensures that sub-tours are avoided. The condition also ensures that a mapping exists between the \textit{TDTSP tour} and a sequence that represents the order of cities in the route. Note that a TDTSP tour \( \Upsilon \) may be perceived as a cycle of all cities, rather than a cycle of all nodes of \( \Gamma \). Denote by \( C_i \), the set of all nodes associated with the city \( i \). Therefore, a \( \Upsilon \) may be perceived as a tour between these clusters, where for each cluster, there is exactly one in-going edge and exactly one out-going edge. Then, the TDTSP could be defined as the problem of finding the shortest TDTSP tour. The next step is to transform
the weighted graph $\Gamma$ to a corresponding weighted graph $G$, in such a way that solving the TSP instance $G$ is equivalent to solving the TDTSP instance $\Gamma$. This transformation is achieved through the use of a useful graph theoretic tool called in-out subgraphs.

Consider a weighted graph $G$, which defines an instance of TSP, and denote by $T^{opt}$ an optimal tour of $G$. Now, suppose that $G$ contains a vertex-induced subgraph $S$. Clearly, $T^{opt}$ must visit the vertices in $S$. Suppose that $T^{opt}$ contains an edge $(u,v)$ such that $u \notin S$ and $v \in S$. In such a case, we say that $v$ is an in-node of $S$ on the tour $T$, and the edge $(u,v)$ is an in-edge of $S$ on the tour $T$. One can define an out-node and out-edge analogously.

**Definition 4.3.2. In-out subgraph**

Let $G$ be a weighted graph corresponding to an instance of TSP. If, for any optimal TSP tour $T^{opt}$ in $G$, the following two conditions are met for a vertex-induced subgraph $S$, we say that $S$ is an in-out subgraph in $G$.

1. There is only one in-edge of $S$ on $T$, and only one out-edge of $S$ on $T$.
   Equivalently, there is only one in-node of $S$ on $T$, and only one out-node of $S$ on $T$.

2. There exists a function $f$ such that, if $i$ is the in-node of $S$ on $T$, then $f(i)$ is the out-node of $S$ on $T$. We refer to $f$ as the in-out function of $S$.

Condition 1 of an in-out subgraph is equivalent to saying that the in-out subgraph is only entered and exited a single time in any TSP optimal tour. This implies that, in the context of TSP, the in-out subgraph functions similarly to a single vertex, in that it can be visited only once. Condition 2 of an in-out subgraph is somewhat more subtle. Although it may be possible to enter or exit an in-out subgraph at many nodes, once a particular in-node is used, condition 2 ensures
that a particular, corresponding out-node is used, according to the in-out function \( f \).

The primary benefit of in-out subgraphs is in modelling constrained TSP, since in-out function \( f \) offers some control over which vertices may be visited next, based on which vertex was visited previously. In-out subgraphs were first described in [48], where the definition given was somewhat more general than used here. In [48], in-out subgraphs were defined in such a way that they must satisfy conditions 1 and 2 for any Hamiltonian cycles rather than the optimal tour. This required the presence of particular structures that ignore the weights of the graph. However, they can be made much smaller through utilizing the weights and focusing on the optimal tour only.

Now, recall that an instance of TDTSP is represented by a graph \( \Gamma \) which contains \( n \) connected components. Corresponding to each city is a cluster \( C_i \) containing \( n \) nodes, where node \([i, t]\) corresponds to city \( i \) at step \( t \). Our objective is to produce a new weighted directed graph, \( G \), such that the optimal TSP tour of \( G = (V, E', w') \) corresponds to an optimal TDTSP tour of \( \Gamma \). Note that \( G \) contains the same vertex set as \( \Gamma \); then it also contains the clusters \( C_i \), for \( i = 1, 2, \ldots, n \). In keeping with the conditions for TDTSP, we want \( G \) to be constructed in such a way that any optimal tour \( T^{opt} \) obeys the following:

- Each cluster \( C_i \) is entered and exited exactly once on \( T^{opt} \).

- If cluster \( C_i \) is entered at time step \( t \), it is exited at time step \( 1 + t \mod n \).
  In other words, if \([i, t]\) is the in-node of \( C_i \) on \( T^{opt} \), then \([i, 1 + t \mod n]\) must be the out-node of \( C_i \) on \( T^{opt} \).

- Each cluster \( C_i \) must be visited consecutively.

The first two requirements can be met by replacing each \( C_i \) with an appropriately constructed in-out subgraph. Specifically, we require an in-out subgraph for which
the function \( f \) maps a vertex \([i, t]\) to the vertex \([i, 1 + t \mod n]\) for all choices of \( t \). However, recall that the definition of in-out subgraphs also depends on the construction of the rest of \( G \) as well. We now indicate how the edges and weights can be chosen for cluster \( C_i \) such that the vertex-induced subgraph of \( C_i \) is an appropriate in-out subgraph.

Suppose that \( P \) is an upper bound for the length of the TDTSP tour of \( \Gamma \). Finding such a value is easy; for example, \( P \) may be obtained by a local search algorithm, nearest neighbor algorithm, or by simply summing the largest edges connected to each node. Then, the edge set \( E' \) consists of two disjoint subsets \( E_1 \) and \( E_2 \), defined along with their weights as follows.

\[ E_1 \text{ edges}: \text{ Set } E_1 = E, \text{ that is, the set of edges from } \Gamma. \text{ Then for each } (u, v) \in E_1, \text{ set } w'(u, v) = P + w(u, v). \]

\[ E_2 \text{ edges}: \text{ } E_2 \text{ contains the edges } ([i, 1 + t \mod n], [i, t]), \forall i, t. \text{ Then for each } (u, v) \in E_2, \text{ set } w'(u, v) = 0. \]

Note that each edge of \( E_2 \) lies inside vertex-induced subgraph of \( C_i \), for a particular value of \( i \). Figure 4.3 gives an example of the vertex-induced subgraph of \( C_i \) for \( n = 5 \). As constructed above, for all values of \( i \), vertex-induced subgraph of \( C_i \) is a directed cycle of size \( n \).

![Figure 4.3: In-out-subgraph of \( C_i \) for \( n = 5 \).](image)

**Lemma 4.3.3.** Consider any vertex-induced subgraph of \( C_i \) of \( G \). Without loss of generality, suppose \([i, t] \in C_i \). Then there exists a unique path in the vertex-
induced subgraph of \( C_i \) from \([i, t]\) to \([i, 1 + t \pmod{n}]\), that includes all nodes of \( C_i \).

Proof. Since the vertex-induced subgraph of \( C_i \) is a cycle, removing the edge \(([i, 1 + t \pmod{n}],[i, t])\) from the subgraph results in a path from \([i, t]\) to \([i, 1 + t \pmod{n}]\) which includes all nodes of the subgraph. It is clear that this is the unique such path. \(\square\)

Assuming \( T \) a TSP tour \( T \) of \( G \) does not contain penalty edges, the length of the TSP tour is \(|T|_G = \sum_{(v,w) \in T} \omega'(v, w)\). Similarly, for a TDTSP tour \( \Upsilon \) of \( \Gamma \), let \(|\Upsilon|_\Gamma = \sum_{(v,w) \in \Upsilon} \omega(v, w)\) be the length of \( \Upsilon \).

Lemma 4.3.4. Consider an instance of TDTSP, \( \Gamma \), and the corresponding instance of TSP, \( G \), defined as above. Then, for any TDTSP tour \( \Upsilon \) of \( \Gamma \), there is a corresponding TSP tour \( T \) of \( G \) such that \(|T|_G = |\Upsilon|_\Gamma + nP\) and \( \Upsilon = T \cap E \).

Proof. The TDTSP tour \( \Upsilon \) contains exactly one edge for each step \( t = 1 \ldots n \). Suppose that \( T \) is constructed as follows. Begin with \( t = 1 \), by adding the first edge from \( \Upsilon \) to \( T \). Without loss of generality, the first edge goes to a vertex \([i, 1]\) for some \( i \). Then, add to \( T \) the edges contained in the path described by Lemma 4.3.3, which covers all nodes of \( C_i \), and ends in \([i, 2]\). Without loss of generality, for some \( j \), the second edge of \( \Upsilon \) is \(([i, 2],[j, 2])\). Then, add the path from \([j, 2]\) to \([j, 3]\) described by Lemma 4.3.3. This process can be continued until for all \( t \), the corresponding edges are added to \( T \). It is clear that in this process the nodes are all visited consecutively, and all nodes are visited. Therefore, \( T \) is a TSP tour of \( G \).

Each of the edges contained within the clusters are \( E_2 \) edges, and so have zero weight. However, the \( n \) edges from \( \Upsilon \) are all \( E_1 \) edges, and by definition they each have weight \( P \) larger than in \( \Gamma \), and so \(|T|_G = |\Upsilon|_\Gamma + nP\). Furthermore, by construction of \( T \), it is evident that \( \Upsilon = T \cap E \). \(\square\)
Lemma 4.3.5. Consider $G$ as defined above. Then for each $i = 1, 2, \ldots, n$, the vertex-induced subgraph of $G$ defined by the nodes in $C_i$ is an in-out subgraph of $G$, and the in-out function of $C_i$ satisfies $f([i, t]) = [i, 1 + t \pmod n]$.

Proof. Consider any optimal tour of $G$, $T^{opt}$. If $T^{opt}$ contains $k$ edges from $E_1$, then by definition $|T^{opt}|_G > kP$. Since there are $n$ clusters, and the only way to travel between clusters is via edges in $E_1$, we know that $k \geq n$. However, we also know from Lemma 4.3.4 that for any TDTSP tour $T$, there is a corresponding tour of $G$ with weight equal to $|T|_\Gamma + nP < (n + 1)P$. Since $T^{opt}$ is optimal, it must also have weight less than $(n + 1)P$, and hence, $k \leq n$. Therefore, we have $k = n$, and so exactly $n$ edges from $E_1$ are used in $T^{opt}$.

Since $T^{opt}$ is a tour, it must visit every cluster, and since exactly $n$ edges from $E_1$ are used, each cluster must be visited exactly once. Hence, the first condition of an in-out subgraph is satisfied.

Next, consider $C_i$, and suppose that $[i, t]$ is the in-node of $C_i$ on $T^{opt}$. Since $C_i$ is visited only once, and $T^{opt}$ is a tour, it must be the case that every node in $C_i$ is visited consecutively. Hence, edges from $E_2$ must be used until every node in $C_i$ is visited. However, for each node $[i, t]$ there is only a single choice available from $E_2$, which is to visit $[i, t \pmod n - 1]$. Hence, it is clear that the final node of $C_i$ that will be visited is $[i, 1 + t \pmod n]$. Therefore, the in-out function of $C_i$ satisfies $f([i, t]) = [i, 1 + t \pmod n]$, completing the proof. □

Figure 4.4 illustrates a simple of a conversion from TDTSP to TSP for $n = 3$. The above results allow us to now state the main theorem of this section.

Theorem 4.3.6. The optimal TSP tour $T^{opt}$ of $G$ corresponds to an optimal TDTSP tour $\Upsilon^*$ of $\Gamma$, where $\Upsilon^* = T^{opt} \cap E_1$.

Proof. Let us first show that $\Upsilon^*$ is a valid TDTSP tour. From Lemma 4.3.5, it is
clear that $T^{opt}$ visits each cluster exactly once so Condition 1 of a TDTSP tour is satisfied. Furthermore, if the cluster is entered at step $t$, it is exited at step $1 + t \pmod{n}$, thus Condition 2 of a TDTSP tour is satisfied. Finally, since $T^{opt}$ is a TSP tour, the nodes are visited consecutively. Since for each city $i$, each cluster $C_i$ has exactly one in-edge and one out-edge, then $C_i$ clusters are visited consecutively for all values of $i$. The in-edges and out-edges are all contained in $\Upsilon^*$, therefore, the cities are visited consecutively and $\Upsilon^*$ satisfies Condition 3 of a TDTSP tour.

All that remains is to show is that $\Upsilon^*$ is optimal. By Lemma 4.3.4, $|T^{opt}|_G = |\Upsilon^*|_\Gamma + nP$. Consider any other TDTSP tour $\Upsilon'$. By Lemma 4.3.4, there exists a TSP tour $T'$ such that $|T'|_G = |\Upsilon'|_\Gamma + nP$. Since $T^{opt}$ is optimal, it follows that,

$$|\Upsilon^*|_\Gamma = |T^{opt}|_G - nP \leq |T'|_G - nP = |\Upsilon'|_\Gamma$$
The inequality above holds for any TDTSP tour $\mathcal{Y}'$, and therefore, $\mathcal{Y}^*$ is an optimal tour of $\Gamma$. □

For the cases where it is difficult to obtain an optimal solution, the following proposition is useful for obtaining heuristic solutions.

**Proposition 4.3.7.** Suppose TSP tours $T_1$ and $T_2$ of $G$ are such $\mathcal{Y}_1 = T_1 \cap E$ and $\mathcal{Y}_2 = T_2 \cap E$ are both TDTSP tours. Then if $|T_1|_G < |T_2|_G$, then $|\mathcal{Y}_1|_\Gamma < |\mathcal{Y}_2|_\Gamma$.

*Proof.* By Lemma 4.3.4, $|T_1|_G = nP + |\mathcal{Y}_1|_\Gamma$ and $|T_2|_G = nP + |\mathcal{Y}_2|_\Gamma$. It immediately follows that $|\mathcal{Y}_1|_\Gamma < |\mathcal{Y}_2|_\Gamma$. Edges of $G$ may be divided into two sets $E_1$ and $E_2$. Since $|T_1|_G < |T_2|_G$, it immediately follows that $|\mathcal{Y}_1|_\Gamma < |\mathcal{Y}_2|_\Gamma$. □

The significance of Proposition 4.3.7 is that if the value of a TSP tour is shorter, the corresponding TDTSP tour is also shorter. This has implications of heuristic solutions to TDTSP because improving a tour on the TSP graph $G$ results in a better TDTSP tour of $\Gamma$, even if the optimal solution is not obtained.

The TSP conversion of TDTSP described above is highly relevant to STSP. First, both $\Gamma$ and $G$ contain $N = n^2$ nodes. The degree of each node is $\sqrt{N}$, resulting in relatively sparse instances. Second, take any TDTSP tour $\mathcal{Y}$ of $\Gamma$. Note that by Lemma 4.3.4, there exists a corresponding TSP tour $T$ in $G$. The construction of $T$ in the proof of Lemma 4.3.4, results in a tour without any gaps. Thus, in order to find the TSP tours that correspond to TDTSP tours, the gaps on the tour should be minimized. Accordingly, the TDTSP is equivalent to the instance of STSP defined on $G$ and the STSP solution methods are applicable.

Appendix A.1 includes a pilot study on randomly generated instances of TDTSP solved using the transformation method. The study included in the appendix, does not incorporate the structural properties of the TDTSP instances. Without incorporating these structural properties, the solvers are significantly disadvent-
taged. At least for Concorde TSP solver, there is a clear way to incorporate some of these structural constraints, which is a topic for future research.

### 4.4 Application 2: DNA-Assembly Problem

Watson and Crick [123] discovered the double-helix structure of DNA, and won the 1962 Nobel Prize in Physiology or Medicine. DNA is the molecule that contains the genetic information; the information that dictates how an organism grows develops and functions. Since the discovery of the DNA structure, enormous multidisciplinary efforts have been dedicated to extracting this information from DNA molecules of various different species. Most notably, the Human Genome Project was a multi-billion dollar project with the participation of multiple countries which sequenced the human genome for the first time [69]. Science Magazine estimated that the total economic impact of DNA research was up to a trillion dollars on the US economy alone [100] by 2013.

To understand the challenges of extracting the genetic information from the DNA, we first discuss the structure of a DNA molecule. DNA consists of two connected strands which form a double-helix structure, and its building blocks are called Nucleobases. The sequence of these nucleobases on the DNA constructs the genetic code, which determines the organic function of proteins in an organism. The primary bases in a DNA molecule are adenine (A), cytosine (C), guanine (G), thymine (T), where on the opposing strands, A always pairs with T, and C pairs with G.

DNA is the largest known molecule in the universe and its width to length ratio poses a severe challenge for extracting the genetic code. The width of the DNA molecule is only 2 nanometres while if the entire human DNA strand was stretched out in a line, its length would reach up to 2 meters. Extracting the information
is currently only possible through first fragmenting of multiple copies of a DNA, and then scanning the information on these DNA fragments. The sequence of bases on a scanned DNA fragment is called a read. Finding the precise sequence of bases using the information from the reads gives rise to a puzzle that is called the de novo assembly. The de novo assembly problem is the problem of aligning and merging fragments of DNA to construct the original DNA sequence without prior knowledge as to the size or structure of the full sequence.

During the early days of DNA sequencing, it quickly became clear that computers are necessary for solving de novo assembly problem, even for sequences that were thousands of times smaller than the human genome [117]. Early algorithms used greedy choices for merging the reads [18, 65]. In particular, reads would be iteratively merged together, where in each iteration the largest available overlap determines the merge choices. In general, while greedy algorithms are fast, they tend to make many non-optimal choices that may significantly affect the quality of a solution.

Currently there are two major solutions methods for the de novo assembly. The first direction is the overlap-layout-consensus (OLC) method and the second method is de Bruijn method [28]. The work of this section is related to the work of Kececioglu and Myers [76] on the OLC method. They proposed a model of the DNA assembly based on the the Shortest Common Superstring problem (SCSP); given a set of strings of letters, find the shortest string such that it is a superstring for all strings in the set. The SCSP is an \( \mathcal{NP} \)-Hard problem and it is equivalent to the traveling salesman problem. They constructed a graph called the overlap graph, where each read is represented by a node. Pairs of reads with significant overlaps are connected to each other by an edge in the overlap graph. Possible alignments of the reads are then well represented by the paths in the overlap graph.
Similar to the previous work on OLC, we also formulate the assembly problem as a variation of the SCSP, where the aim is to reconstruct the DNA sequence in a way that maximizes the overlaps of the fragments. We propose a variation of the overlap graph proposed in Kececioglu and Myers [76] which we call the TSP overlap graph. There are two main differences between our approach and the approach of [76]. One is the representation of each read in the construction of the overlap which leads to a more accurate representation, and second is our solution method which attacks the problem more directly.

### 4.4.1 Notation

We first introduce a few concepts and useful notation, relevant to the super string problem. Consider two strings $S_1$ and $S_2$ where there exists a suffix of $S_1$ that is equal to a prefix of $S_2$. The largest suffix of $S_1$ that is equal to a prefix of $S_2$ is called the overlap of $S_1$ and $S_2$ and it is denoted by $(S_1 \cap S_2)$. For example, for $S_1 = abcd eff$ and $S_2 = deffaac$, the overlap of $S_1$ and $S_2$ is $(S_1 \cap S_2) = def$. Note that overlap is not commutative, that is $(S_1 \cap S_2) \neq (S_2 \cap S_1)$.

The remainder of $S_2$ after $(S_1 \cap S_2)$ is the suffix of $S_2$ which is obtained by removing the prefix $(S_1 \cap S_2)$ of $S_2$. In the example above, the remainder of $S_2$ after $(S_1, S_2)$ is the string $aac$. Then the merge operation $\text{merge}(S_1, S_2)$ concatenates string $S_1$ to the remainder of $S_2$ after $(S_1 \cap S_2)$, such that the letters of $S_1$ appear first. If $S_1 = abcd eff$ and $S_2 = deffaac$, then $\text{merge}(S_1, S_2) = abcd effaac$.

Consider a sequence of strings $(S_1, S_2, \ldots, S_n)$. We say that the sequence of strings $(S_1, S_2, \ldots, S_n)$ is equivalent to the string $S$, if

$$S = \text{merge}(\ldots(\text{merge}(\text{merge}(S_1, S_2), S_3), \ldots), S_n).$$

The length of a string $S$ is denoted by $\text{len}(S)$. The length of a sequence $(S_1, S_2,$
..., $S_n$) is defined to be $\text{len}(\mathcal{S})$ where $\mathcal{S} \equiv (S_1, S_2, \ldots, S_n)$. We have

$$\text{len}(\mathcal{S}) = \text{len}(S_1) + \sum_{i=2}^{n} \left[ \text{len}(S_i) - \text{len}(S_{i-1} \cap S_i) \right]$$

$$= \sum_{i=1}^{n} \text{len}(S_i) - \sum_{i=2}^{n} \text{len}(S_{i-1} \cap S_i). \tag{4.2}$$

Let us move on to notations that are specifically related to DNA-assembly. A **read** is string of DNA letters $A$, $C$, $G$ and $T$. Since DNA has two complementary strands, a fragment may come from either strands. Let us denote the reverse complement of read $\mathcal{R}$ by $\tilde{\mathcal{R}}$. For the DNA letters, $\tilde{A} = T$, $\tilde{T} = A$, $\tilde{C} = G$ and $\tilde{G} = C$. Also for example, if a read $\mathcal{R} = CAGGT$, then $\tilde{\mathcal{R}} = C\tilde{A}G\tilde{G}T = ACCTG$.

The existence of the overlap $(\mathcal{R}_1 \cap \mathcal{R}_2)$ forces the existence of the overlap $(\tilde{\mathcal{R}}_2 \cap \tilde{\mathcal{R}}_1)$.

Accordingly, there are four independent ways that a pair of reads can overlap, namely, $(\mathcal{R}_1 \cap \mathcal{R}_2)$, $(\mathcal{R}_2 \cap \mathcal{R}_1)$, $(\mathcal{R}_1 \cap \tilde{\mathcal{R}}_2)$ and $(\tilde{\mathcal{R}}_2 \cap \mathcal{R}_1)$.

Consider a set of $n$ DNA reads, $\mathcal{R}_1, \ldots, \mathcal{R}_n$. We make the assumption that no read can be fully contained in another read. This is standard practice because such a read is almost always redundant in DNA-assembly so it is disregarded during a pro-processing phase. Suppose the overlaps between each pair of read is calculated (see [76, 112] for methods of efficiently calculating the overlaps).

A reconstructed DNA sequence may be represented as sequence of reads, where either the read or its reverse complement is part of the sequence. For example, for $\mathcal{R}_1 = AACT$, $\mathcal{R}_2 = ACTG$, $\mathcal{R}_3 = TCAG = \tilde{CTGA}$ and $\mathcal{R}_4 = GGCC$, the sequence $(\mathcal{R}_1, \mathcal{R}_2, \tilde{\mathcal{R}}_3, \mathcal{R}_4)$ is equivalent to the super-string (DNA sequence) $\mathcal{S} = AACTGA'GGCC$. Note that there are no overlaps between $\tilde{\mathcal{R}}_3$ and $\mathcal{R}_4$, and this gap is marked by the ($'$) symbol (the symbol is merely a marker and it is not part of the string).
4.4.2 TSP Formulation of the Overlaps Graph

The representation of the original DNA sequence indicates the order of letters in one of the two strands. On the other hand, the data from fragments could come from either fragment. When a read represents a fragment of one strand, its reverse complement represents a fragment in the opposing strand. Therefore, in DNA assembly, in addition to the order of fragments in the original DNA, the orientation of each fragment also needs to be determined. This issue slightly differentiates the DNA-assembly problem from the classical formulation of the SCSP.

In the overlap graph, Kececioglu and Myers [76] represented each read as a node, which does not take into account the orientation of each fragment. The orientation of each fragment is determined by a solving the “fragment orientation problem”, an NP-complete problem that is solved by an approximation algorithm. In this section, we avoid the overlap problem altogether by representing each read as a subgraph, rather than a single node, and the the orientation of the read is integrated in the subgraph.

The shortest common superstring problem for the DNA fragments may be formally formulated as follows. Given $n$ reads $\mathcal{R}_1, \ldots, \mathcal{R}_n$, find a sequence of the $n$ reads, where for all $i$ either $\mathcal{R}_i$ or $\tilde{\mathcal{R}}_i$ is a member of the sequence, and its corresponding super-string has the minimal length. For short, we refer to this formulation as the DNA-assembly Problem. Note that this formulation is the portion of the de novo assembly after the read data is cleaned and corrected (see [77, 112, 80] for literature on cleaning the data and correcting reads). By 4.2, we see that in order to minimize the length of the super-string for a set of given reads, the sum the overlaps for pairs of consecutive reads must be maximized.

We now construct a constrained asymmetric TSP on a graph $G$ for which its solu-
tion may be mapped to the shortest common super string of the DNA fragments. For any read $R_i$, two nodes $p_i$ and $s_i$ in $G$ represent the left end and the right end of the read of $R_i$ respectively. Let $V_1 = \bigcup_{i=1}^{n} \{p_i, s_i\}$ and let

$$E_1 = \bigcup_{i=1}^{n} \{(p_i, s_i), (s_i, p_i)\},$$

be a set of edges with zero weight. Edges of $E_1$ connect the pair of nodes that correspond to every read. We refer to the edges in $E_1$ as orientation edges.

**Table 4.4:** Orientation edges of $E_1$ for each $i = 1, 2, \ldots, n$. The Representation column indicates the meaning of a directed edge if selected in a tour.

<table>
<thead>
<tr>
<th>Edge</th>
<th>Weight</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(p_i, s_i)$</td>
<td>0</td>
<td>$R_i$ appears in its given orientation</td>
</tr>
<tr>
<td>$(s_i, p_i)$</td>
<td>0</td>
<td>$R_i$ appears in its reverse complement orientation</td>
</tr>
</tbody>
</table>

Now let us construct the edges that correspond to the remainders of a reads after overlaps. Consider any pairs of reads $R_i$ and $R_j$. As mentioned before, there are four possible ways $R_i$ and $R_j$ can overlap. Also since the orientation of reads is not known, there are two possible ways that each overlap is oriented. Table 4.5 describes the corresponding two edges in $G$ representing each direction of an overlap in the four possible cases. We call these edges overlap edges. The set of all overlap edges is denoted by $E_2$.

Finally, we add a starting node $v_0$ to have a clear reference for the starting location where $v_0$ is connected to all other nodes with bi-directional edges. The set of edges connected to $v_0$ is

$$E_o = \bigcup_{u \in V_1} \{(v_0, u), (u, v_0)\}.$$

Each $u \in V_1$ is associated with a read, say $R_i$. Then the weight $(v_0, u)$ is $\text{len}(R_i)$ and the weight of $(u, v_0)$ is zero.

Given the set of reads, and the list of overlaps between reads, the weighted directed graph $G = (V, E, \omega)$ is well defined for $V = \{v_0\} \cup V_1$ and $E = E_0 \cup E_1 \cup E_2$ where
The intention of this discussion is to define a constrained TSP problem on $G$ that corresponds to the DNA-assembly problem. Therefore, let us narrow our focus to special tours of $G$ defined below.

**Definition 4.4.1. Qualified tour**

Let $T$ be a tour of $G$ such that for each $i = 1, 2, \ldots n$, either $(p_i, s_i) \in T$ or $(s_i, p_i) \in T$. Then tour $T$ is called a qualified tour of $G$.

### Table 4.5: Overlap edges for the four possible ways that $R_i$ and $R_j$ may overlap. The Representation column indicates the string that corresponds to the directed edge.

<table>
<thead>
<tr>
<th>Type</th>
<th>Overlap</th>
<th>Edge</th>
<th>Weight</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$(R_i, R_j)$</td>
<td>$(s_i, p_j)$</td>
<td>$\text{len}(R_j) - \text{len}(R_i)$</td>
<td>Remainder of $R_j$ after $(R_i, R_j)$</td>
</tr>
<tr>
<td></td>
<td>$(\bar{R}_i, R_j)$</td>
<td>$(p_j, s_i)$</td>
<td>$\text{len}(R_i) - \text{len}(R_j)$</td>
<td>Remainder of $\bar{R}_i$ after $(\bar{R}_j, R_j)$</td>
</tr>
<tr>
<td>2</td>
<td>$(R_j, R_i)$</td>
<td>$(s_j, p_i)$</td>
<td>$\text{len}(R_i) - \text{len}(R_j)$</td>
<td>Remainder of $R_i$ after $(R_j, R_i)$</td>
</tr>
<tr>
<td></td>
<td>$(\bar{R}_j, \bar{R}_i)$</td>
<td>$(p_i, s_j)$</td>
<td>$\text{len}(R_i) - \text{len}(R_j)$</td>
<td>Remainder of $\bar{R}_j$ after $(\bar{R}_i, \bar{R}_j)$</td>
</tr>
<tr>
<td>3</td>
<td>$(R_i, \bar{R}_j)$</td>
<td>$(s_i, s_j)$</td>
<td>$\text{len}(R_j) - \text{len}(\bar{R}_i, R_j)$</td>
<td>Remainder of $\bar{R}_j$ after $(R_i, \bar{R}_j)$</td>
</tr>
<tr>
<td></td>
<td>$(\bar{R}_i, \bar{R}_j)$</td>
<td>$(s_j, s_i)$</td>
<td>$\text{len}(R_i) - \text{len}(\bar{R}_j, R_i)$</td>
<td>Remainder of $\bar{R}_i$ after $(\bar{R}_j, \bar{R}_i)$</td>
</tr>
<tr>
<td>4</td>
<td>$(\bar{R}_j, R_i)$</td>
<td>$(p_j, p_i)$</td>
<td>$\text{len}(R_i) - \text{len}(\bar{R}_j, R_i)$</td>
<td>Remainder of $R_i$ after $(\bar{R}_j, R_i)$</td>
</tr>
<tr>
<td></td>
<td>$(\bar{R}_i, R_j)$</td>
<td>$(p_i, p_j)$</td>
<td>$\text{len}(R_j) - \text{len}(\bar{R}_j, R_i)$</td>
<td>Remainder of $R_j$ after $(\bar{R}_i, R_j)$</td>
</tr>
</tbody>
</table>

### Table 4.6: Edges of $E_0$ for $i = 1, 2, \ldots n$. The Representation column indicates the meaning of a directed edge if selected in a tour.

<table>
<thead>
<tr>
<th>Edge</th>
<th>Weight</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(v_0, p_i)$</td>
<td>$\text{len}(R_i)$</td>
<td>First read is $R_i$</td>
</tr>
<tr>
<td>$(v_0, s_i)$</td>
<td>$\text{len}(\bar{R}_i)$</td>
<td>First read is $\bar{R}_i$</td>
</tr>
<tr>
<td>$(p_i, v_0)$</td>
<td>0</td>
<td>Termination of the read sequence</td>
</tr>
<tr>
<td>$(s_i, v_0)$</td>
<td>0</td>
<td>Termination of the read sequence</td>
</tr>
</tbody>
</table>

$\omega : E \rightarrow \mathbb{R}$ is the function that maps each edge to its weight as described in Tables 4.4–4.6. Figure 4.5 illustrates the construction of $G$ for a simple example involving five reads. In this example, overlaps containing less than two letters are discarded.

The intention of this discussion is to define a constrained TSP problem on $G$ that corresponds to the DNA-assembly problem. Therefore, let us narrow our focus to special tours of $G$ defined below.
Let us represent the directed cycle made from the edges of $T$ by the sequence $q_T = (v_0, a_1, \ldots, a_{2n}, v_0)$, where $a_k$ is the $k$-th node in the path. A qualified tour of $G$ has the following property.

**Lemma 4.4.2.** Suppose $T$ is a qualified tour of $G$. Let the sequence $q_T = (v_0, a_1, \ldots, a_{2n}, v_0)$ be the directed cycle made from edges of $T$. Then $(a_{2k−1}, a_{2k}) \in E_1$ for all $k = 1, 2, \ldots, n$.

**Proof.** Edges $(p_i, s_i)$ or $(s_i, p_i)$ are both orientation edges. Since, excluding $v_0$, every nodes of $G$ is either $p_i$ or $s_i$, for some $i$. Therefore, with the exception of $v_0$, every node on the qualified tour $T$ is connected to exactly one orientation edge. Then, we use induction to prove the lemma. Without loss of generality, $a_1 = p_i$ or $a_1 = s_i$ for some $i$. In either case, $(v_0, a_1)$ is not an orientation edge, therefore,
\((a_1, a_2)\) must be an orientation edge, or else the assumption that \(T\) is a qualified tour is violated. So for \(k = 1\), the edge \((a_{2k-1}, a_{2k})\) is an orientation edge.

For the inductive step of the proof we need to show that if \((a_{2k-1}, a_{2k})\) is an orientation edge, then \((a_{2k+1}, a_{2k+2})\) is also an orientation edge. Since both \(a_{2k-1}\) and \(a_{2k}\) correspond to the the same read, say \(R_i\), then the \(a_{2k+1}\) which comes right after \(a_{2k}\) must correspond to a different read, say \(R_j\). Since, for \(a_{2k+1}\), the edge \((a_{2k}, a_{2k+1})\) is not an orientation edge, then by the same argument as before, \((a_{2k+1}, a_{2k+2})\) must be an orientation edge. \(\square\)

Lemma 4.4.2 enables us to create a mapping from any qualified tour to a read sequence below.

**Definition 4.4.3. Read sequence of \(T\)**

Suppose \(T\) is a qualified tour. Let the sequence \(q_T = (v_0, a_1, \ldots, a_{2n}, v_0)\) be the directed cycle made from edges of \(T\). Then the *read sequence of \(T\)* is constructed as follows. For \(k = 1, 2, \ldots n\), iteratively add the following to the sequence

(i) if for some \(i\), \((a_{2k-1}, a_{2k}) = (p_i, s_i)\), add \(R_i\) to the sequence,

(ii) if for some \(i\), \((a_{2k-1}, a_{2k}) = (s_i, p_i)\), add \(\tilde{R}_i\) to the sequence.

Note that the choices in the iterative steps are consistent with the representation of edges described in Table 4.4. Furthermore, by Lemma 4.4.2, in cases (i) and (ii) are exhaustive and its iteration adds a read or the reverse complement of a read. To emphasize the property that the sequence of \(T\) is well defined for any qualified tour \(T\), we state the following lemma.

**Lemma 4.4.4.** Let \(T\) be any qualified tour. There exist a sequence of reads that is the read sequence of \(T\).
Let us illustrate an example of a read sequence of a tour $T$ of the Figure 4.5. For the tour $T = \{(v_0, p_1), (p_1, s_1), (s_1, p_2), (p_2, s_2), (s_2, s_3), (s_3, p_3), (p_3, s_4), (s_4, p_4), (p_4, p_5), (p_5, s_5), (s_5, v_0)\}$, the directed cycle may be represented by $q_T = (v_0, p_1, s_1, p_2, s_2, s_3, p_3, s_4, p_4, p_5, s_5, v_0)$. The read sequence of $T$ is $(R_1, R_2, \tilde{R}_3, \tilde{R}_4, R_5) \equiv ATTGGGCTACGCTAA$. This example is visualized in Figure 4.6.

The three main properties of the graph $G$ are discussed below. The first is that all DNA super-strings made from the reads are well-represented in the graph.

**Lemma 4.4.5.** Consider a DNA-assemble problem with reads $R_1, R_2, \ldots, R_n$. Let $S$ be a DNA super-string made from the reads. Then there exist a qualified tour $T$ in $G$ such that the read sequence of $T$ is equivalent to $S$.

**Proof.** Without loss of generality, suppose $S \equiv (S_1, S_2, \ldots, S_n)$ such that for each $i = 1, 2, \ldots, n$, there exist a unique $k$ such that either $S_k = R_i$ or $S_k = \tilde{R}_i$. Construct tour $T$ as follows:

1. add $v_0$
for $k = 1, 2, \ldots, n$, find $i$ such that either $S_k = R_i$ or $S_k = \tilde{R}_i$

2.1) if $S_k = R_i$, first add $p_i$, then add $s_i$.

2.2) else (if $S_k = \tilde{R}_i$), first add $s_i$, then add $p_i$.

(3) add $v_0$.

By construction, $T$ is a qualified tour and the read sequence of $T$ is $(S_1, S_2, \ldots, S_n) \equiv S$. □

The second property concerns the length of a qualified tour.

**Lemma 4.4.6.** Consider a qualified tour $T$ of $G$. The value of $T$ is equal to the length of the read sequence of $T$.

**Proof.** Without loss of generality, the read sequence of $T$ is $(S_1, S_2, \ldots, S_n)$ such that for each $i = 1, 2, \ldots, n$, there exist a unique $k$ such that either $S_k = R_i$ or $S_k = \tilde{R}_i$. By equation (4.2) the length of the string $S \equiv (S_1, S_2, \ldots, S_n)$ is

$$
\text{len}(S) = \sum_{i=k}^{n} \text{len}(S_k) - \sum_{k=2}^{n} \text{len}(S_{k-1} \cap S_k).
$$

The sum of the weights of $T \cup E_0$ is $\text{len}(S_1)$. The sum of the edges of $T \cup E_1$ is zero. Finally, by the construction of edges of $E_2$ in $G$ the sum of the edges of $T \cup E_2$ is $\sum_{k=2}^{n} [\text{len}(S_k) - \text{len}(S_{k-1})]$. Since $E_0$, $E_1$ and $E_2$ cover all edges of $T$, then the total cost of the tour $T$ is equal to the length of string $\text{len}(S)$. □

The third and most significant property of the graph $G$ is as follows.

**Theorem 4.4.7.** Consider graph $G$ constructed from a set of reads. Suppose $T^{\text{opt}}$ is the shortest qualified tour of $G$. The read sequence of $T^{\text{opt}}$ is the shortest common DNA super-string.

**Proof.** By Lemma 4.4.4, any tour may be mapped to a read sequence of $T$. By
Lemma 4.4.6, the length of the sequence is equal to the value of $T$. It follows that, the read sequence of $T^{\text{opt}}$ is shorter or equal in length to any read sequence of another qualified tour $T$. By Lemma 4.4.5, for any DNA-super string $S$ made from the reads, there exist a qualified tour $T$ in $G$ such that the read sequence of $T$ is equivalent to $S$. Since such a tour $T$ is at least as long as $T^{\text{opt}}$, then the read sequence of $T^{\text{opt}}$ is at least as long as the read sequence of $T$. Therefore, the read sequence of $T^{\text{opt}}$ must be the shortest common DNA super-string. □

Theorem 4.4.7 shows that the shortest common DNA super-string problem is equivalent to the asymmetric traveling salesman problem on qualified tours. The constraint that must be added to the TSP is that for all $i = 1, 2, \ldots, n$, either $(p_i, s_i) \in T$ or $(s_i, p_i) \in T$.

For many TSP solvers, like Concorde and LKH, there are in-built features to easily implement these constraints. However, to accommodate for all TSP solvers, the constrained asymmetric TSP may easily be transformed to an (un-constrained) asymmetric TSP by adding a middle node between $p_i$ and $s_i$. Specifically, for $i = 1, 2, \ldots, n$, introduce a middle node $m_i$ and replace $(p_i, s_i)$ with $(p_i, m_i)$ and $(m_i, p_i)$, and replace $(s_i, p_i)$ with $(m_i, s_i)$ and $(s_i, m_i)$. The middle node forces a tour to travel either from $p_i$ to $s_i$ through $m_i$, or from $s_i$ to $p_i$ through $m_i$. The former is equivalent to $(p_i, s_i) \in T$ in $G$ and the latter is equivalent to $(s_i, p_i) \in T$ in $G$.

### 4.4.3 The Case for a STSP Formulation

Graph $G$ produced in the last subsection is a complete graph. In this sub-section, we make a few minor alterations to the graph $G$ to create a sparse graph. There are two main reasons for making these changes. First reason is that some properties of the DNA-assembly problem are better reflected in a sparse graph. The second reason is reducing the amount of memory required for storing the weights.
Let us elaborate on the first reason. The depth of coverage of a DNA-assembly data is the average number of times a letter (i.e., a base) of the DNA string appears in the data (set of reads). Assuming the depth of coverage is large enough and the reads cover bases with sufficient span throughout the DNA sequence, there must be significant portions of some pairs of reads overlap, which come from nearby sections of the DNA, and the number of gaps in the DNA should be minimal. Without sufficient depth of coverage, the results of the DNA-assembly are not meaningful. Therefore, it is reasonable to assume that there are significant portions of reads that overlap with some other reads.

Furthermore, in the DNA-assembly problem, only meaningful (sufficiently large) overlaps should be considered. For example, consider a read $R_i$. The probability that $R_i$ has an overlap of size 2 with another, entirely random, read $R_j$ is $1/16$ (if they both have at least two bases). Such an overlap is not of interest in the context of the DNA-assembly problem because it may simply reflect a coincidence rather than the relative positioning of $R_i$ and $R_j$ in the DNA string. When dealing with millions of reads, merging $R_i$ and $R_j$ only with an overlap of size 2 is not justified because the probability that this overlap is coincidental is extremely likely.

In fact, the reason that DNA-assembly problem may be formulated by a variation of the super-string problem is that it generally only reasonable to merge two reads when the overlap is large enough so that a coincidental overlap is extremely unlikely. An appropriate criteria for filtering meaningful overlaps may be developed. For example the SGA assembler dismisses overlaps that are smaller than the 75 bases of the read [112]. The consequence of this filter is the significant sparsification of the problem as for any read there will be much more limited choices for merging.

Furthermore, an effective method for gathering the DNA-assembly data seeks to minimize the uncovered gaps. Thorough coverage of the entire DNA sequence is an
important factor for measuring the effectiveness of the data collection. Therefore, for data with sufficient quality, it is also reasonable to prioritize minimizing the gaps in the DNA string.

The second benefit from considering a sparse graph is making the problem more memory efficient. While TSP instances with millions of nodes have been solved to near-optimal solutions, almost all of these instances do not require the storage of all weights\(^3\). Storing all the weights for a large full-graph may lead to enormous requirements for memory. Therefore, it modifying graph \(G\) to a sparse graph provides a more practical option for solving large examples.

To this end, we now construct graph \(G'\), which is a modification of graph \(G\) by sparsification. The sparse graph \(G'\) is equal to the graph \(G\) without the edges associated with insignificant overlaps. Note that constructing \(G'\) does not require prior construction of \(G\). Instead, \(G'\) may be constructed by the same process as the construction of \(G\), except, without adding edges in \(E_2\) for overlaps that are not significant.

### 4.4.4 Computational Study on Simulated Instances

In this subsection we investigate the following questions. Is a TSP model a computationally viable method for solving the DNA-assembly problem?

A TSP formulation of the DNA-assembly problem is often dismissed because of the \(\mathcal{NP}\)-Hard nature of the TSP. In Nature Biotechnology, Compeau, P. E et al. [28] examines a TSP formulation,\(^4\) and for solving problems with millions of reads states that “the computational burden [of the TSP approach] was so

---

\(^3\)For example, for solving Euclidean instances of TSP, an algorithm only requires the storage of coordinates for each node, and the distances are calculated as required. This demands only \(O(n)\) memory compared to \(O(n^2)\) for a complete graph.

\(^4\)The TSP formulation in [28] is distinct from the formulation in this chapter.
large that most New Generation Sequencing projects have abandoned the [...] approach.”

Finding a conclusive answer for the question of whether a TSP approach could be useful for applications on real DNA sequencing data is a challenging task. There are many difficulties in dealing with real DNA sequencing data, such as correcting the errors in the data, and efficiently calculating overlaps. These components require careful design, implementation and integration which is beyond the scope of this thesis. Instead, to evaluate the potential viability of this approach, we have designed a simple pilot study using computer generated examples.

To get around the complexity of creating an overlap from real DNA-sequencing data, we have attempted to simulate the structure of a TSP overlap graph instead. Consequently, the computer generated examples reflect some of the complexities involved in the DNA-assembly using a TSP formulation. In particular, we are interested in discovering to what extent the \( \mathcal{NP} \)-Hard nature of the TSP constitutes a real obstacle for using the TSP model of DNA-assembly. The computational complexity of the DNA assembly is frequently stated as the major reason for avoiding this direction of research. In our experiment, a TSP overlap graph is generated directly for \( n = 50k, 100k, 250k, 500k \), reads of size 1000 bases. The vertices \( V \), edges of \( E_0 \), and edges of \( E_1 \) are forced by the selection of \( n \). The overlap edges of \( E_2 \) are simulated by randomly generating edges with random weights of size up to 900, that is at overlaps of least 100 base pairs. Furthermore, the edges are generated such that each node is incident to 5 edges of \( E_2 \). Considering there are two nodes that represent each read, the edges of \( E_2 \) represent 10 possible overlaps for each read.

The results of sLKH are compared to a greedy algorithm for the TSP, since the greedy approximation algorithms have been one of the primary methods used for solving overlap graphs [18, 65]. The simple greedy algorithm benchmarked against
sLKH starts with the first node, and makes the greedy choice at each step until all reads are exhausted. We performed all of the tests of this chapter on a single core of an Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz processor, on a UNIX OS server. Parameters of Table 4.7 were used for this experiment. Parameters INITIAL_PERIOD, MAX_TRIALS and RUNS were fixed to limit the computational time, and ASCENT_CANDIDATES was capped for disregarding the penalty edges in the initial stage.

Table 4.8 shows that the results of sLKH is highly effective in closing the gaps on these instances. While the naive greedy algorithm produces around 1 gap per 8 reads, sLKH produced around 1 gap per 100,000 reads. It is worth noting that for all instances of up to 100k reads, the instances are solved to a single reference sequence without any gaps. The total length of the reference sequence by the sLKH algorithm was consistently around one-third shorter than the greedy algorithm, which is a substantial improvement. Furthermore, the use of the sparse input format in sLKH was proved to be very useful. Remarkably, the virtual memory requirement for sLKH process for all instance was under 100MB.

The pre-processing of a TSP overlap graph almost always significantly reduces the number of nodes by efficiently pre-solving the trivial choices. For example, there are usually many reads in the graph for which there is only one way to merge them with another read. Solving the trivial cases may decrease the number of nodes to a fraction of the initial size. The Table above indicates that it is computationally practical to solve TSP overlaps created from hundreds of thousands
Table 4.8: Performance of sLKH on simulated instances of TSP overlap compared to a greedy algorithm. The generated graphs simulate the structure of the overlap TSP graph with reads of 1000 base pairs and overlaps of at least 100 base pairs. Column Length represents the length of the reference sequence generated by the algorithm and the number of gaps in the reference sequence is denoted by $g(T)$. The percentage of improvement in term of length by sLKH compared to the greedy algorithm is reported in the Impv. column.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Reads</th>
<th>Greedy</th>
<th>sLKH</th>
<th>Time (s)</th>
<th>Impv.(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Length</td>
<td>$g(T)$</td>
<td>Length</td>
<td>$g(T)$</td>
</tr>
<tr>
<td>Ovl-25k-a</td>
<td>25k</td>
<td>8243884</td>
<td>3120</td>
<td>5600588</td>
<td>0</td>
</tr>
<tr>
<td>Ovl-25k-b</td>
<td>25k</td>
<td>8245864</td>
<td>3092</td>
<td>5641295</td>
<td>0</td>
</tr>
<tr>
<td>Ovl-25k-c</td>
<td>25k</td>
<td>8296729</td>
<td>3053</td>
<td>5651792</td>
<td>0</td>
</tr>
<tr>
<td>Ovl-25k-d</td>
<td>25k</td>
<td>8253204</td>
<td>3071</td>
<td>5660562</td>
<td>0</td>
</tr>
<tr>
<td>Ovl-50k-a</td>
<td>50k</td>
<td>16526320</td>
<td>6156</td>
<td>11300102</td>
<td>0</td>
</tr>
<tr>
<td>Ovl-50k-b</td>
<td>50k</td>
<td>16428763</td>
<td>6103</td>
<td>11261981</td>
<td>0</td>
</tr>
<tr>
<td>Ovl-50k-c</td>
<td>50k</td>
<td>16489007</td>
<td>6236</td>
<td>11245387</td>
<td>0</td>
</tr>
<tr>
<td>Ovl-50k-d</td>
<td>50k</td>
<td>16374182</td>
<td>6111</td>
<td>11209550</td>
<td>0</td>
</tr>
<tr>
<td>Ovl-100k-a</td>
<td>100k</td>
<td>32857726</td>
<td>12361</td>
<td>22489102</td>
<td>0</td>
</tr>
<tr>
<td>Ovl-100k-b</td>
<td>100k</td>
<td>32926705</td>
<td>12347</td>
<td>22441278</td>
<td>0</td>
</tr>
<tr>
<td>Ovl-100k-c</td>
<td>100k</td>
<td>32973677</td>
<td>12186</td>
<td>22579841</td>
<td>0</td>
</tr>
<tr>
<td>Ovl-100k-d</td>
<td>100k</td>
<td>32811669</td>
<td>12372</td>
<td>22457909</td>
<td>0</td>
</tr>
<tr>
<td>Ovl-250k-a</td>
<td>250k</td>
<td>82557585</td>
<td>31056</td>
<td>56332830</td>
<td>3</td>
</tr>
<tr>
<td>Ovl-250k-b</td>
<td>250k</td>
<td>82207659</td>
<td>30868</td>
<td>56134019</td>
<td>2</td>
</tr>
<tr>
<td>Ovl-250k-c</td>
<td>250k</td>
<td>82217849</td>
<td>30627</td>
<td>56327246</td>
<td>3</td>
</tr>
<tr>
<td>Ovl-250k-d</td>
<td>250k</td>
<td>82370900</td>
<td>30774</td>
<td>56225393</td>
<td>2</td>
</tr>
<tr>
<td>Ovl-500k-a</td>
<td>500k</td>
<td>164766819</td>
<td>61308</td>
<td>112470682</td>
<td>6</td>
</tr>
<tr>
<td>Ovl-500k-b</td>
<td>500k</td>
<td>164989479</td>
<td>61532</td>
<td>112689075</td>
<td>6</td>
</tr>
<tr>
<td>Ovl-500k-c</td>
<td>500k</td>
<td>164891314</td>
<td>61701</td>
<td>112548954</td>
<td>5</td>
</tr>
<tr>
<td>Ovl-500k-d</td>
<td>500k</td>
<td>164715315</td>
<td>61748</td>
<td>112498297</td>
<td>5</td>
</tr>
<tr>
<td>Ovl-1M-a</td>
<td>1M</td>
<td>329318314</td>
<td>123067</td>
<td>225304924</td>
<td>11</td>
</tr>
<tr>
<td>Ovl-1M-b</td>
<td>1M</td>
<td>329667847</td>
<td>123427</td>
<td>225114301</td>
<td>15</td>
</tr>
<tr>
<td>Ovl-1M-c</td>
<td>1M</td>
<td>328939164</td>
<td>123009</td>
<td>225136661</td>
<td>11</td>
</tr>
<tr>
<td>Ovl-1M-d</td>
<td>1M</td>
<td>329505176</td>
<td>123421</td>
<td>225003872</td>
<td>12</td>
</tr>
</tbody>
</table>
of reads. This is likely to translate to solving instances with potentially millions of reads when a pre-solver is included. Furthermore, the ability of the TSP approach may be further extended by one or two orders of magnitudes if combined with clustering methods that break down the problem into smaller sub-problems [9]. The conducted experiment suggests that it is feasible for a well-implemented TSP approach to obtain solutions to problems with millions of reads, that are significantly superior to the solutions obtained by greedy algorithms.

The third generation sequencing technology, that uses biological nanopores to extract the DNA data, is capable of long reads containing tens of thousands of reads [111]. As a result, the number of reads required to obtain sequencing data for a complete human genome is within the bound of a few million reads. This pilot study demonstrates that a TSP model is potentially practical for problems of such size, and therefore points to a rewarding direction of research for the full implantation of a assembly software based on the TSP approach for assisting the state-of-the-art technology.
Chapter 5

The Clustered Generalized Traveling Salesman Problem

5.1 Introduction

In this chapter, we consider an important variant of the classical traveling salesman problem (TSP) called the clustered generalized traveling salesman problem (CGTSP). Recall that the definition of the TSP is as follows: given a set of nodes and the length of travel between pairs of nodes, find the route of the minimal length that visits each node exactly once and comes back to the initial node. Then, suppose the nodes are partitioned into sets called clusters, and each cluster is further partitioned into sets of nodes called subclusters. Recall Definition 1.4.3 that the CGTSP is defined as follows:

Find the minimal length route that satisfies the following:

(1) the route visits exactly one node per subcluster, and,

(2) within each cluster, all of the visited nodes are visited consecutively.

There are two primary motivations for studying CGTSP. First, CGTSP is one of the most generalized variations of TSP and it encompasses many important
problems. We can consider the classical TSP to be a special case of CGTSP where each node constitutes a separate cluster. Similarly, well-known related variations of TSP, namely, the generalized traveling salesman problem (GTSP) and the clustered traveling salesman problem (CTSP) are also special cases of CGTSP; for the former, a single subcluster fully occupies a cluster, and for the latter, each subcluster contains precisely one node. Figure 5.1 illustrates GTSP, CTSP and CGTSP. Both GTSP and CTSP are single-layer expansions of the TSP where nodes are only partitioned into groups. Then, the problem in the GTSP is to find a minimum length tour that includes exactly one node from each group, whereas the salesman in the CTSP must visit all nodes of a group before moving to a new group. Figures 5.1(a) and 5.1(b) show tours for these problems. However, as Figure 5.1(c) shows, CGTSP is a double-layer expansion of TSP in which the external layer of the problem is a CTSP. That is to say, upon visiting a cluster, we must visit each subcluster contained therein before moving to a new cluster. In the internal layer, each cluster independently generates an instance of GTSP where the salesman must visit exactly one node of each subcluster.

Variations of the TSP have been studied extensively in the literature (e.g., see Gutin and Punnen [53]). In particular, applications of GTSP and CTSP are discussed in detail in [81] and [82], respectively. The algorithmic solutions to GTSP include exact algorithms [41], approximation algorithms [47], local search heuristics [73, 74, 113] and meta-heuristics [20, 6]. Similarly, examples of reported ap-

![Figure 5.1: The linkages between GTSP, CTSP and CGTSP.](image)

(a) An example tour of GTSP  (b) An example tour of CTSP  (c) An example tour of CGTSP
approaches for solving CTSP include exact algorithms [5], approximation algorithms [12], local search heuristics [38, 94] and meta-heuristics [83, 23].

The literature also includes studies on variations of TSP that are closely related to CGTSP. For instance, Zhang et al. [127] introduced the Tabu clustered traveling salesman problem (TCTSP), and proposed two meta-heuristics to solve it. In a TCTSP, the nodes are partitioned into two types of clusters: GTSP-like clusters and CTSP-like clusters. The salesman visits exactly one node of a cluster if it is the first type, whereas he must consecutively visit all nodes within each cluster if it is the second type. In [17], the authors considered a problem called the family traveling salesman problem (FTSP) where the nodes are partitioned into cluster, which they refer to as families. Specifically, the FTSP requires the salesman to visit a predefined number of nodes in each family consecutively. The predefined number for each family is at least one and at most equal to the number of nodes in the family. A cluster is GTSP-like if the predefined number is equal to one, whereas it is CTSP-like if the number is equal to the number of nodes in the cluster. Accordingly, the FTSP can also be seen as an extension of GTSP and CTSP. Note that TCTSP is a special case of FTSP in that it is equivalent to a FTSP where the predefined number for each particular cluster is equal to either one or the number of nodes in the cluster. However, CGTSP has a different hierarchical structure that means it cannot be seen as a special case of FTSP. More precisely, the nodes in FTSP are partitioned into clusters, whereas CGTSP goes one step further by partitioning the nodes in each cluster into subclusters.

The second motivation for studying CGTSP is its close alignment with potential applications in modern logistics. Extensive work has been done to model traditional logistics systems in the context of the TSP (e.g. see [7]). The wider scope of CGTSP offers a more powerful alternative for modern logistics challenges. For instance, technologies such as radio frequency identification (RFID) allow for item localization as an indoor logistics activity. Thus, similar items may be stored sepa-
rately in different locations in an automated storage and retrieval system (ASRS). It can be argued that a CGTSP model is a more suitable framework compared to the a TSP model.

In addition to ASRS, technologies such as an unmanned aerial vehicle (UAV), commonly known as a drone, allow collaboration between a delivery truck and drones. Thus, customers may be partitioned into subclusters in a drone-assisted parcel delivery service (PDS). This means that delivery and pick-up of a set of orders can be modeled as a CGTSP where the truck needs to visit only one of the customers from each subcluster, as drones visit the rest.

One common method for solving any variation of the TSP is to convert the problem to the classical TSP. For example, this is the case for both GTSP [14, 16, 48] and CTSP [25]. While the idea of transformation is natural, in reality, the resultant TSP instances may have a difficult structure to the point where existing TSP solvers do not offer high-quality solutions. Considering this critical factor in the transformation, Helsgaun [64, 63] evaluated the performance of his state-of-the-art TSP solver Lin-Kernighan heuristic (LKH) for transformed GTSP and CTSP instances. LKH was used as a black box optimization algorithm without fundamental modifications to the TSP heuristic method, and was successful at producing competitive results for both CTSP and GTSP. These studies illustrate that TSP solvers could be highly effective in solving variations of TSP with the aid of an appropriate transformation methods.

The outline of this chapter is as follows. In Section 5.2, we develop an algorithm which transforms CGTSP to the classical TSP. The transformation allows us to take advantage of the vast literature for algorithmic solutions to TSP (e.g. [7, 62, 10, 9]). The proposed transformation technique has the advantage that its framework is dynamically aggregated and can be assumed as a potential candidate for transforming other cluster-based variations of the TSP into classical
TSP (e.g. to enable the transformation of TCTSP or FTSP into TSP). In Section 5.3, we report on a computational study on randomly-generated CGTSP instances which compares the transformation method to an alternative method in the literature, and demonstrate that the transformation method produces more competitive results compare to the existing algorithms. Finally, given this new ability to efficiently solve CGTSP instances, we discuss potential applications of CGTSP in logistics. Each application is accompanied by a computational study on application-inspired CGTSP instances, in Section 5.4. In the computational studies, we have exploited two powerful TSP solvers Concorde [7] and LKH [62] to generate high-quality CGTSP solutions in a reasonable time.

5.2 Transformation of CGTSP into TSP

We present a linear-time transformation from CGTSP to TSP where the number of nodes remains unchanged. We first introduce the notation adopted throughout the section, and then we discuss the details of the transformation. We prove the validity of the transformation and conclude that the computational complexity of CGTSP and TSP are equivalent. An algorithmic description of the transformation is included at the end of this section.

5.2.1 Notation

We first discuss a graph representation of a CGTSP instance. Consider the (directed or undirected) graph $\Gamma = (V, \mathcal{E}, \omega)$ where $V$ is a set of nodes, $\mathcal{E}$ is a set of edges representing the routes between the pairs of nodes. Furthermore, $\omega : \mathcal{E} \to \mathbb{R}$ is a function that maps pairs of nodes in $\mathcal{E}$ to their corresponding distance. The set of nodes $V$ is divided into a set of $N_C$ disjoint clusters, i.e. $V = \bigcup_{c=0}^{N_C-1} C_c$. Each cluster $C_c$ is further divided into $NS(c)$ disjoint subclusters,
that is \( C_c = \bigcup_{s=1}^{NS(c)} S_{c,s} \). We define \( NV(c,s) := |S_{c,s}| \), and hence the number of nodes is \( n = |V| = \sum_{c=0}^{N_C-1} \sum_{s=1}^{NS(c)} NV(c,s) \). We also define \( N_S \) to be the total number of subclusters, which is equal to \( \sum_{c=0}^{N_C-1} NS(c) \).

A node \( v = [c,s,i] \), is the \( i \)-th node of subcluster \( S_{c,s} \) of cluster \( C_c \). We assume that \( v_0 \) is the fixed starting node of the CGTSP instance, and it is the sole node in subcluster \( S_{0,1} \) and cluster \( C_0 \). Any edge \((u,v) \in E\) represents the directed link from \( u \) to \( v \), and its length is represented by \( \omega(u,v) \). The situation is illustrated, for example, in Table 5.1 for a CGTSP with three clusters \( C_0, C_1 \) and \( C_2 \), and five subclusters \( S_{0,1}, S_{1,1}, S_{1,2}, S_{2,1} \) and \( S_{2,2} \). The length of any particular edge \( \omega(u,v) \) may represent any metric of interest such as distance of travel, cost of moving or the time of movement from one node to another node. Furthermore, inner links for each subcluster \( S_{c,s}, \forall c \in \{0, 1, 2\} \) and \( s \in \{1, 2\} \), are forbidden while outer links for the subcluster are given in the matrix of Table 5.1.

An instance of CGTSP is clearly defined by \((\Gamma, C, S)\), where \( C \) is the set of all clusters, \( S \) is the set of all subclusters. Where there is no ambiguity about \( C \) and \( S \), we use only \( \Gamma \) as an abbreviated representation of a CGTSP instance.

The formal definition of CGTSP that we will use in this chapter is as follows. The CGTSP requests the path of the shortest distance from \( v_0 \) which visits exactly one node of each subcluster, visits all subclusters of each cluster consecutively, and finally returns to the starting node \( v_0 \). Figure 5.2 illustrates a CGTSP with five clusters and a feasible tour of the CGTSP that travels through 18 nodes out of 78 nodes. In contrast to a TSP where a tour is a permutation of all nodes, a \( CGTSP\)-tour contains exactly one node for each subcluster. More precisely, a sequence of nodes \( \Upsilon = (v_0, \ldots, v_0) \) represents a CGTSP-tour if the following conditions are satisfied:

**Condition 1:** The selection of nodes is in a sequence, starting from \( v_0 \) and ending in \( v_0 \).
**Condition 2:** Exactly one node is selected from each subcluster.

**Condition 3:** Without loss of generality, let \( v \) be the \( i \)-th element of the sequence \( \Upsilon \), and suppose \( v \) is inside cluster \( C_c \). Then, either the \((i + 1)\)-th element is also in \( C_c \), or \( v \) is the last occurrence of a node in cluster \( C_c \) for the remainder of the sequence \( \Upsilon \). In other words, each cluster of nodes is exited (and entered) exactly once.

A CGTSP-tour \( \Upsilon \) is considered to be *feasible* if for any consecutive pair of nodes \( u \) and \( v \) in \( \Upsilon \), the pair \((u, v)\) is a valid edge of \( E \).

![Figure 5.2](image.png)

*Figure 5.2: A feasible tour of the CGTSP with five clusters.*

Analogous to a CGTSP-tour, we represent a TSP tour by a sequence of nodes. Unlike a CGTSP-tour that covers only some of the nodes, a TSP tour \( T = (v_0, \ldots, v_0) \), includes all nodes of the graph and represents the order of nodes on the TSP tour. We also use a matrix representation of a tour \( X_T = [x_{u,v}] \), where the entry \( x_{u,v} = 1 \) if the edge \((u, v)\) is on the TSP tour, and \( x_{u,v} = 0 \) otherwise. Table 5.2 provides a summary of the various notations we will use in this chapter.
<table>
<thead>
<tr>
<th>$C_0$</th>
<th>$S_{(0,1)}$</th>
<th>$S_{(1,1)}$</th>
<th>$C_1$</th>
<th>$S_{(1,2)}$</th>
<th>$C_2$</th>
<th>$S_{(2,2)}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>

Table 5.1: A matrix of weights of edges for a CGTSP with clusters $C_0$, $C_1$, $C_2$, and subclusters $S_{0,1}$, $S_{1,1}$, $S_{1,2}$, $S_{2,1}$, $S_{2,2}$
Table 5.2: List of notations used in Chapter 5.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>number of nodes, i.e., $n =</td>
</tr>
<tr>
<td>$N_C$</td>
<td>number of clusters</td>
</tr>
<tr>
<td>$C_c$</td>
<td>set of nodes of the $c$-th cluster</td>
</tr>
<tr>
<td>$NS(c)$</td>
<td>number of subclusters in the $c$-th cluster</td>
</tr>
<tr>
<td>$N_S$</td>
<td>total number of subclusters, i.e., $\sum_{c=0}^{N_C-1} NS(c)$</td>
</tr>
<tr>
<td>$S_{c,s}$</td>
<td>set of nodes in the $s$-th subcluster of the $c$-th cluster</td>
</tr>
<tr>
<td>$NV(c,s)$</td>
<td>number of nodes in the $s$-th subcluster of the $c$-th cluster, i.e., $NV(c,s) =</td>
</tr>
<tr>
<td>$v_0$</td>
<td>fixed initial node of a CGTSP-tour</td>
</tr>
<tr>
<td>$S_{0,1}, C_0$</td>
<td>subcluster and cluster of the sole node $v_0$ respectively, i.e., $S_{0,1} = C_0 = {v_0}$</td>
</tr>
<tr>
<td>$v = [c, s, i]$</td>
<td>node $v$, the $i$-th node of the $s$-th subcluster of the $c$-th cluster</td>
</tr>
<tr>
<td>$V$</td>
<td>set of nodes of CGTSP, i.e., $V = {v_0, v_1, \ldots, v_{n-1}}$</td>
</tr>
<tr>
<td>$E$</td>
<td>set of all edges of CGTSP, i.e., $E = {(u,v) : u, v \in V, u \neq v}$</td>
</tr>
<tr>
<td>$\Gamma$</td>
<td>graph $(V,E,\omega)$ representing a CGTSP</td>
</tr>
<tr>
<td>$G$</td>
<td>graph of the constrained TSP where $G = (V,E,\omega_G)$</td>
</tr>
<tr>
<td>$G'$</td>
<td>graph of the TSP where $G' = (V,E,\omega'_G)$ and penalty weights are $P_1$ and $P_2$</td>
</tr>
<tr>
<td>$\Upsilon$</td>
<td>sequence representation of a CGTSP-tour in the graph $\Gamma$</td>
</tr>
<tr>
<td>$T$</td>
<td>sequence representation of an eligible tour (Definition 5.2.1)</td>
</tr>
<tr>
<td>$</td>
<td>T</td>
</tr>
<tr>
<td>$</td>
<td>T</td>
</tr>
<tr>
<td>$X_T = [x_{u,v}]$</td>
<td>matrix representation of a tour $T$ with binary values</td>
</tr>
<tr>
<td>$\omega(u,v)$</td>
<td>length of the edge $(u,v)$</td>
</tr>
</tbody>
</table>

5.2.2 Transformation of CGTSP to a Constrained TSP

In order to transform a CGTSP instance $\Gamma$ into a TSP instance $G'$ we need the following. First we need a method of constructing $G'$ and the updated distance function from the original instance. In addition, we need a function $f$ which maps solutions of $G'$ into solutions of $\Gamma$. Finally, we need to prove that $f$ maps any optimal solution of $G'$ to an optimal solution of $\Gamma$. As an intermediate step, we define a constrained TSP on a graph $G$ and show that the constrained TSP on $G$ is equivalent to the CGTSP. Later, we transform the constrained TSP on $G$ to the (unconstrained) TSP on $G'$. 
Let $G = (V, E, \omega_G)$ be a weighted directed graph, where $V$ is the same set of nodes as in the graph $\Gamma$. The set of edges $E$ and their associated weights are constructed as follows. For all subclusters $S_{c,s}$, the $s$-th subcluster of the $c$-th cluster, add the following edges to $E$:

- Directed edges $([c, s, NV(c, s)], [c, s, 1])$ with weight 0.
- Directed edges $([c, s, i], [c, s, i + 1])$ for $i = 1, \ldots, NV(c, s) - 1$, with weight 0.
- Directed edges $(u, v)$ between $u = [c, s, i]$ and $v$, for all $u \in S_{c,s}$ and $v \notin S_{c,s}$. If $i \neq 1$, the weight of the edge is $\omega([c, s, i - 1], v)$. Otherwise, for $i = 1$, the weight of the edge is $\omega([c, s, NV(c, s)], v)$.

The above construction has two important properties that will be used later. First, for any node $u = [c, s, i]$ in a subcluster $S_{c,s}$, there is a unique node $v \in S_{c,s}$ such that $(u, v)$ is a directed edge. In other words, in a feasible tour $T$ the node that follows $u$ is either the unique node $v$, or it is not in $S_{c,s}$. Second, any edge $(u, v)$ where $u \in S_{c,s}$ but $v \notin S_{c,s}$, corresponds to the edge $(p, v)$ in the graph $\Gamma$, where $p = [c, s, i - 1]$ if $i \neq 1$, and $p = [c, s, NV(c, s)]$ otherwise.

The TSP induced by the graph $G$ is unambiguous. Now let us apply the following constraints to the TSP induced by $G$, and later show that the constrained TSP is equivalent to the CGTSP.

**Definition 5.2.1. Eligible TSP tour**

An *eligible TSP tour* of the graph $G$ is any feasible TSP tour $X_T = [x_{u,v}]$ satisfying the following conditions:

$$\sum_{u \in S_{c,s}} \sum_{v \notin S_{c,s}} x_{u,v} = 1, \quad \text{for } \forall S_{c,s},$$

(5.1)
\[ \sum_{u \in C_c} \sum_{v \not\in C_c} x_{u,v} = 1, \quad \text{for } \forall C_c. \quad (5.2) \]

For a tour \( T \), an \textit{in-node of a cluster} is a node in the cluster which occurs immediately after a node outside of the cluster in \( T \). Also, an \textit{out-node of a cluster} is a node in the cluster which occurs immediately before a node outside of the cluster in the sequence \( T \). The terms \textit{in-node of a subcluster} and \textit{out-node of a subcluster} are defined analogously for subclusters. Respectively, constraints (5.1) and (5.2) are equivalent to the conditions that the eligible tour has exactly one out-node for each subcluster and one out-node for each cluster. Since \( T \) is a tour, there is exactly one corresponding in-node for every out-node of \( T \). Then, each cluster and subcluster of an eligible tour has exactly one in-node as well.

We can now define a function \( f \) that maps an eligible tour of \( G \) to a CGTSP-tour of \( \Gamma \).

**Lemma 5.2.2.** Let the function \( f(T) = T' \) that maps an eligible tour \( T \) to a sequence \( T' \) be defined as follows:

- \textit{Start with} \( v_0 \) \textit{in the sequence.}
- \textit{Iterate through the elements of the tour that are not} \( v_0 \) \textit{and apply the following operation for each element. If the current element is an in-node, add it to the sequence. Otherwise, skip it.}
- \textit{Finally add} \( v_0 \) \textit{to the end of the sequence.}

Then the sequence \( T' \) represents a CGTSP-tour.

**Proof.** We need to show that the sequence \( T' \) satisfies Conditions 1, 2 and 3 for a CGTSP-tour. It is trivial that Condition 1 is satisfied. Then, by the definition of an eligible tour, there is only one in-node for every subcluster in \( T \), the iterative
step would select exactly one node per subcluster in $T'$, and hence Condition 2 is satisfied. Finally, since there is only one in-node for each cluster in the eligible tour $T$, all of the nodes in the cluster are placed consecutively in $T$. Therefore, the iterative step places all of the nodes in a particular cluster consecutively in $T'$, satisfying Condition 3. In summary, $T'$ satisfies the three conditions, and it is, therefore, a CGTSP-tour.

It should be emphasized that Lemma 5.2.2 allows us to redefine the range of function $f$ as the space of feasible CGTSP-tours of $\Gamma$.

**Lemma 5.2.3.** The function $f(T) = \Upsilon$, which maps $T$, an eligible tour $T$ of $G$, to $\Upsilon$, a feasible CGTSP-tour of $\Gamma$, is bijective.

**Proof.** We first show that $f$ is injective. Consider any subcluster $S$ of the graph. Without loss of generality, the eligible tour $T = (v_0, \ldots, v_{\text{in}}, \ldots, v_{\text{out}}, \ldots, v_0)$, where $v_{\text{in}}$ and $v_{\text{out}}$ are in-nodes and out-nodes of a subcluster $S$ respectively and the subsequence $Q_S = (v_{\text{in}}, \ldots, v_{\text{out}})$ of $T$, includes all of the nodes in $S$. Under the function $f$, the sub-sequence $Q_S$ in $T$ collapses to the single element $(v_{\text{in}})$ in $\Upsilon = (v_0, \ldots, v_{\text{in}}, \ldots, v_0)$.

Since $Q_S$ is a sub-sequence of a feasible TSP tour, there exists an edge between consecutive elements of $Q_S$. Moreover, the construction of the graph $G$ ensures that for any node in $S$, there is only a unique directed edge to another node in $S$. Then, since consecutive nodes in the sub-sequence $Q_S$ represent edges between nodes in $S$, what follows after $v_{\text{in}}$ in the sub-sequence $Q_S$ is forced and therefore unique. Then, the collapse of $Q_S$ to $v_{\text{in}}$ is injective. It follows that $f$ is an injective function.

To show that $f$ is onto, we should find an eligible tour $T$ for any feasible CGTSP-tour $\Upsilon$ such that $f(T) = \Upsilon$. Recall that $\Upsilon$ starts and ends with $v_0$ (Condition 1), contains exactly one in-node per subcluster (Condition 2), and the nodes of the
cluster in $\Upsilon$ are placed consecutively (Condition 3) for any cluster. We construct $T$ as follows:

**Step 1:** Add $(v_0)$ to $T$.

**Step 2:** Denote the next element of $\Upsilon$ that is not already added to $T$ by $v_{in}$. Also denote the subcluster that contains $v_{in}$ by $S$. Add $v_{in}$ to $T$. If the added element is $v_0$, return $T$.

**Step 3:** Denote the last node added to $T$ by $v$. Construction of $G$ ensures that there exists a unique node $v \in S$ where $(u, v)$ is a directed edge in $G$. If $v \neq v_{in}$, add $v$ to $T$ and repeat step 3. Otherwise, go to step 2.

It is clear that $f(T) = \Upsilon$. All we need to show is that $T$ is in the domain of $f$, i.e., it is an eligible tour. Due to the placement of edges between edges in a subcluster in $G$, step 3 is repeated until all nodes in a subcluster are added to $T$. Therefore, the only out-node of the subcluster added is the the last added node, and $T$ satisfies constraints (5.1). Furthermore, because nodes that belong to the same cluster are placed consecutively in $\Upsilon$, and step 3 only add nodes in the same subcluster (and hence the same cluster), the nodes belonging to the same cluster are also placed consecutively in $T$. So, $T$ also satisfies constraints (5.2) and $T$ is in the domain of $f$. Hence, $f$ is onto, injective, and therefore, bijective. □

Since $f$ is bijective, the function $f^{-1}$ is well-defined. Below, we show that the function $f$ preserves the length of the tour, that is the length of TSP tour $T$ is equal to the length of the CGTSP-tour $\Upsilon$.

**Lemma 5.2.4.** The total length of a feasible CGTSP-tour $\Upsilon$ in $\Gamma$, is equal to the length of the corresponding TSP tour $T = f^{-1}(\Upsilon)$ in $G$.

**Proof.** Let $(u, v)$ be an edge on the CGTSP-tour. Without loss of generality, $\Upsilon = (\ldots, u, v, \ldots)$. Let $S_{c,a}$ be the subcluster of $u$. Then $v \notin S_{c,a}$. Also $T =$
(..., u, ..., u_{out}, v, ...), where \( u_{out} \) is the out-node of the subcluster \( S_{c,s} \) in \( T \).

Note that the edges of \( G \) are designed in a way that the path \((u, ..., u_{out})\) has zero length, and the length of \((u_{out}, v)\) in \( G \) is equal to \( \omega(u, v) \) in \( \Gamma \). Therefore, the length of the path \((u, ..., u_{out}, v)\) in \( G \) is equal to \( \omega(u, v) \) in \( \Gamma \). For any edge \((u, v)\) in \( \Upsilon \), the function \( f^{-1} \) replaces the edge by the path of the same length \((u, ..., u_{out}, v)\) in \( T \). Hence, the length of \( T \) in \( G \) is equal to the length of the tour \( \Upsilon \) in \( \Gamma \). \( \square \)

The following theorem sums up the results of the above discussion.

**Theorem 5.2.5.** The optimal solution of the CGTSP defined on \( \Gamma \) is equivalent to the optimal solution of the instance of TSP defined on \( G \) and constrained by constraints (5.1) and (5.2).

**Proof.** A TSP tour in \( G \) satisfying constraints (5.1) and (5.2) is an eligible tour. By Lemmas 5.2.2–5.2.4, there exists a bijective function \( f \) that maps an eligible tour of \( G \) to a CGTSP-tour of \( \Gamma \), while preserving the length of the tour. Therefore, \( f \) maps the optimal eligible tour of \( G \) to the optimal CGTSP-tour of \( \Gamma \). \( \square \)

### 5.2.3 Transformation to a Classical (Unconstrained) TSP

Solving a constrained TSP, such as the constrained TSP defined on \( G \) in the previous subsection, may be problematic depending on the specifics of the chosen solver; some solvers can accommodate additional constraints to a TSP, while others can not. In order to enable any TSP solver to solve CGTSP, we present a conversion of the constrained TSP introduced in the previous section to a classical TSP, that is, TSP without additional constraints.

First, we partition the edges of \( G \) into three mutually exclusive sets \( E_1, E_2, \) and
Type 1: edge \((u, v)\) is a type 1 edge if \(u, v \in S_{c,s}\) for some fixed \(c\) and \(s\). In other words, the end-nodes of a type 1 edge are in the same subcluster, and we say that \((u, v) \in E_1\).

Type 2: edge \((u, v)\) is a type 2 edge if \(u \in S_{s,c} \subset C_c\) and \(v \in C_c\) but \(v \notin S_{s,c}\). In other words, the end-nodes of a type 2 edge are in the same cluster, but not in the same subcluster, and we say that \((u, v) \in E_2\).

Type 3: edge \((u, v)\) is a type 3 edge if it is not a type 1 or type 2 edge. In other words, the end-nodes of a type 3 edge are not in the same cluster, and we say that \((u, v) \in E_3\).

Let \(\omega_G(u, v)\) denote the length of the edge \((u, v)\) in \(G\). Let \(P_1\) and \(P_2\) be positive constants representing penalty values. Construct the weighted directed graph \(G' = (V, E, \omega'_G)\) as follows. The set of nodes \(V\) and the set of edges \(E\) are identical to the corresponding sets of \(G\) but the weight function \(\omega'_G\) is not identical to \(\omega_G\). Specifically, for each \((u, v) \in E\), edge \((u, v)\) and has the following weight in \(G'\):

- If \((u, v) \in E_1\), then \(\omega'_G(u, v) = \omega_G(u, v)\).
- If \((u, v) \in E_2\), then \(\omega'_G(u, v) = \omega_G(u, v) + P_1\).
- If \((u, v) \in E_3\), then \(\omega'_G(u, v) = \omega_G(u, v) + P_1 + P_2\).

Figure 5.3 illustrates an example of a TSP tour in \(G'\). In this figure, the solid lines represent the edges with non-zero lengths, and the dashed lines represent the dummy edges with length zero.

Note that an eligible tour \(T\) of \(G\) is an eligible tour of \(G'\) as well. The total length of \(T\) in \(G\) and \(G'\) is denoted by \(|T|_G\) and \(|T|_{G'}\), respectively.
Figure 5.3: TSP tour of $G'$ made by transforming the CGTSP graph $\Gamma$ of Figure 5.2.

Lemma 5.2.6. Let $T$ be an eligible tour of $G$. The length of the eligible tour $T$ in $G'$ is

$$|T|_{G'} = N_C P_2 + N_S P_1 + |T|_G.$$

Proof. Since the clusters, subclusters and the edges of $G$ and $G'$ are identical, $T$ is an eligible tour of $G'$. An eligible tour has exactly $N_C$ edges from $E_3$ and $N_S - N_C$ edges from $E_2$. Therefore, the sum of penalty values added to $|T|_G$ is $N_C P_2 + N_S P_1$. □

Lemma 5.2.6 shows that the TSP constrained by the eligible tour constraints on $G$ and $G'$ are equivalent in the sense that, an eligible solution of $G$ is an eligible solution of $G'$, and the values of such a solution solution differs by a constant in $G$ and $G'$.

Corollary 5.2.7. Let $T_1$ and $T_2$ be two eligible tours of $G$. Then, $|T_1|_G \leq |T_2|_G$ if and only if $|T_1|_{G'} \leq |T_2|_{G'}$.

Below, we show that it is possible to choose $P_1$ and $P_2$ such that the eligible tour constraints (5.1) and (5.2) are redundant.
Lemma 5.2.8. Let $T$ be an eligible tour of $G$ and let $M$ be a constant value such that $|T|_G < M$. Construct $G'$ with $P_1 = M$ and $P_2 = NSM$. Also, let $T'$ be a TSP tour of $G'$ such that $T'$ is not an eligible tour. Then, $|T|_{G'} < |T'|_{G'}$.

Proof. By Lemma 5.2.6,

$$|T|_{G'} = NCNSM + NSM + |T|_G$$

$$< (NCNS + NS + 1)M$$

$$\leq (NCNS + 2NS)M.$$ 

Let $X' = [x'_{u,v}]$ be the matrix representation of the tour $T'$, such that if $(u, v)$ is an edge on the tour, $x'_{u,v} = 1$, and $x'_{u,v} = 0$ otherwise. Then,

$$|T'|_{G'} = \sum_{(u,v)\in E_3} P_2 + \sum_{(u,v)\in E_2\cup E_3} P_1 + \sum_{(u,v)\in E} x'_{u,v}\omega(u, v).$$

The number of edges of $T'$ that are also contained in $E_2 \cup E_3$ must be greater than the number of edges of $T$ that are also contained in $E_2 \cup E_3$. This is because an eligible tour has the minimal number of type 2 and type 3 edges by design, one per subcluster (constraints (5.1)) and one per cluster (constraints (5.2)). If $T'$ violates constraints (5.1), there is at least one extra edge from $E_2 \cup E_3$ in $T'$ compared to $T$. So, the number of edges from $E_2 \cup E_3$ in $T'$ is at least $NS + 1$. It follows that,

$$|T'|_{G'} = \sum_{(u,v)\in E_3} NSM + \sum_{(u,v)\in E_2\cup E_3} M + \sum_{(u,v)\in E} x'_{u,v}\omega(u, v)$$

$$\geq NCNSM + (NS + 1)M + \sum_{(u,v)\in E} x'_{u,v}\omega(u, v)$$

$$> |T|_{G'}.$$ 

Similarly, if $T'$ violates constraints (5.2), there is at least one extra type 3 edge in $T'$ compared to $T$. So the number of type 3 edges in $T'$ is at least $NC + 1$. We
have,

\[ |T'|_{G'} = \sum_{(u,v) \in E_3} N_S M + \sum_{(u,v) \in E_2 \cup E_3} M + \sum_{(u,v) \in E} x'_{u,v} \omega(u, v) \geq (N_C + 1) N_S M + N_S M + \sum_{(u,v) \in E} x'_{u,v} \omega(u, v) \geq (N_C N_S + 2 N_S) + \sum_{(u,v) \in E} x'_{u,v} \omega(u, v) > |T|_{G'} . \]

These results enable us to propose the main theorem of this section.

**Theorem 5.2.9.** Let \( M \) be an strict upper bound for the length of the optimal tour of the CGTSP defined by \( \Gamma \). Let the penalty edges of \( G' \) be \( P_1 = M \) and \( P_2 = N_S M \). Suppose the optimal tour of the TSP induced by \( G' \) is \( T^* \). Then \( \Upsilon^* = f(T^*) \) is an optimal CGTSP-tour of \( \Gamma \).

**Proof.** By Theorem 5.2.5, \( M \) is also an upper bound for the optimal eligible tour of \( G \). Therefore, there exists an eligible tour \( T \), such that \( |T|_G < M \). By Lemma 5.2.8, if \( T^* \) is not an eligible tour, then \( |T|_{G'} < |T^*|_{G'} \), which is a contradiction. Therefore, \( T^* \) must be an eligible tour. Take any non-optimal eligible tour \( T \) in \( G \). By Corollary 5.2.7, \( |T^*|_{G'} < |T|_{G'} \) implies that \( |T^*|_G < |T|_G \). Thus, \( T^* \) is also an optimal eligible tour for the TSP induced by \( G \). It then follows, by Theorem 5.2.5, that \( \Upsilon^* = f(T^*) \) is an optimal CGTSP-tour of \( \Gamma \). \[ \square \]

It should be emphasized that both penalty values \( P_1 = M \) and \( P_2 = N_S M \) are loose in the sense that it is possible to select much tighter values for which Theorem 5.2.9 remains true. However, in the writing of this chapter, we have prioritized the concision of the central argument.
The number of steps in the transformation defined on $\Gamma = (V, \mathcal{E}, \omega)$ to a TSP defined on $G' = (V, E, \omega'_G)$ and the transformation from a TSP solution to a CGTSP solution are bounded by $O(|\mathcal{E}|)$ and $O(|V|)$, respectively. It follows that the transformation algorithm is linear-time with respect to the number of edges and implies the following result.

**Corollary 5.2.10.** The computational complexity of CGTSP is equivalent to the computational complexity of TSP.

Note that the formulations of TSP and CGTSP used in this thesis are $\mathcal{NP}$-hard, because there are no known polynomial-time algorithms that could verify optimality of a solution. Corollary 5.2.10 conveys that the complexity of finding an optimal tour for CGTSP is equivalent to the complexity of finding an optimal tour for TSP.

Due to the fact that TSP is a special case of CGTSP where clusters only contain one node, it is trivial that CGTSP is $\mathcal{NP}$-Hard. However, the key point is that $\mathcal{NP}$-Hardness of a problem only implies that the problem is at least as hard as the hardest problems in $\mathcal{NP}$. Corollary 5.2.10 implies that CGTSP, despite its more general formulation, is *not* computationally more difficult than the TSP.

We have presented an algorithmic description of the transformation of a CGTSP to a TSP as Algorithm 3. The transformation of the solution of TSP to the solution of CGTSP is also outlined in Algorithm 4. In this chapter, we have assumed that all CGTSP-tours starts from a fixed node $v_0$, a realistic assumption that also helps with the flow of our theoretical arguments. However, it is easy to modify Algorithm 4 to work for a CGTSP without a fixed starting point. Simply, start from the first in-node of the first cluster, rather than the fixed starting point $v_0$. The transformation Algorithm 3 is valid with or without a fixed starting location.
Algorithm 3: Transformation of a CGTSP to a TSP

Inputs: CGTSP graph $\Gamma = (V, E, \omega)$, the length $\omega(u, v)$ for all $(u, v) \in E$, appropriate penalty values $P_1$ and $P_2$

Outputs: The TSP graph $G' = (V, E, \omega'_G)$

Initialization: for $\forall (u, v) \in E$ allocate memory to $\omega'_G(u, v)$;

/* The set of nodes $V$ contains the information about the clusters and the subclusters of $\Gamma$. */

1 for $c = 0 \ldots N_C - 1$
   2 for $s = 1 \ldots NS(c)$ do
      3   /* Adding the edges to nodes in the subcluster $S_{c,s}$. */
      4      foreach $u = [c, s, i] \in S_{c,s}$ and $i \neq NV(c, s)$ do
      5          $v = [c, s, i + 1]$;
      6          $\omega'_G(u, v) = 0$
      7      end
      8      $u = [c, s, NV(c, s)]$;
      9      $v = [c, s, 1]$;
     10      $\omega'_G(u, v) = 0$;
      /* Adding the edges to nodes outside of the subcluster $S_{c,s}$. */
     11      foreach $u = [c, s, i] \in S_{c,s}$ and $i \neq 1$
     12          foreach $u \in E$ and $v \notin S_{c,s}$ do
     13              if $([c, s, i - 1], v) \in E$ then
     14                  if $v \in C_c$ then
     15                     $\omega'_G(u, v) = P_1 + \omega([c, s, i - 1], v)$;
     16                  end
     17                  else
     18                     $\omega'_G(u, v) = P_2 + P_1 + \omega([c, s, i - 1], v)$;
     19                  end
     20              end
     21          end
     22      end
     23   end
     24 end
     25 return Outputs;
Algorithm 4: Transformation of an eligible TSP tour to a CGTSP-tour

Inputs : Eligible TSP tour $T$
Outputs: CGTSP-tour $\Upsilon$

1. **Initialization**: $\Upsilon = \emptyset$
2. add $v_0$ to $\Upsilon$
3. add $T[1]$ to $\Upsilon$
4. last_subcluster = subcluster of $T[1]$
5. for $\text{index} = 2 \ldots n$ do
   6. Current_Subcluster = subcluster of $T[\text{index}]$
   7. if $\text{current_subcluster} \neq \text{last_subcluster}$ then
      8. add $T[\text{index}]$ to $\Upsilon$
      9. last_subcluster = current_subcluster
   end
6. end
7. return $\Upsilon$

5.3 Comparative Performance on Randomly Generated Instances

Now that we possess a method of transforming CGTSP to TSP, we seek to compare the performance of TSP algorithms on the transformed instances, to the existing approaches to solving CGTSP. The investigation of CGTSP is a relatively recent research effort in the literature, and algorithmic solutions to CGTSP have not yet been widely studied. Arguably, the state-of-the-art algorithmic effort in this field is due to Foumani et al [43] in which both an IP formulation of CGTSP, and a cross-entropy (CE) metaheuristic approach to solving CGTSP were described. We will compare both exact and heuristic solutions. For exact solutions, we use Concorde [7] to solve the transformed TSP instances, and compare it to solutions of the IP formulation provided by CPLEX (v12.5.0.0). For heuristic solutions, we use LKH [62] to solve the transformed TSP instances, and compare it to the CE approach. After conducting a set of pilot experiments, we found that the LKH parameters given in Table 5.3 provide a reasonable balance between quality and solution time.
Table 5.3: Recommended LKH parameters for solving the transformed CGTSP.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX_CANDIDATES</td>
<td>25 to 50</td>
</tr>
<tr>
<td>MAX_TRIALS</td>
<td>$n$ if $n &lt; 10,000$, and 10,000 otherwise</td>
</tr>
<tr>
<td>RUNS</td>
<td>1</td>
</tr>
<tr>
<td>ASCENT_CANDIDATES</td>
<td>500</td>
</tr>
<tr>
<td>All other parameters</td>
<td>default</td>
</tr>
</tbody>
</table>

For exact solutions, the main factor is whether or not the optimal solution is obtained and verified within the determined computational limitations. In contrast, the quality of a heuristic solution is typically determined by comparing it to a lower bound value. However, currently, there is no method for obtaining tight lower bounds for CGTSP. For this reason, we instead compare the heuristic solutions to a relatively trivial solution, obtained by a CGTSP nearest neighbor (NN) algorithm that provides an upper bound value. The nearest-neighbor difference ($NN\ diff$) is obtained by dividing the NN value by the heuristic solution and presented as a percentage value.

In Tables 5.4–5.5, we compare the performance of finding exact and heuristic solutions for the TSP transformation method to exact and heuristic solutions for the IP formulation of CGTSP. In the first table, the performance of Concorde TSP solver and CPLEX are compared as the exact solvers, and in the second table, the performance of LKH and CE metaheuristic are contrasted as the heuristic solvers. The instances of Tables 5.4–5.5 consist of randomly generated vertices in a $1000 \times 1000$ 2D Euclidean space. We use the generic name "\textit{rnd}\textless N_C\textgreater \textless N_S\textgreater \textless N_V\textgreater n\textquotedblright", where $N_C$ is the number of clusters, $N_S$ is the number of subclusters in each cluster, and $N_V$ is the number of vertices in each cluster. For each size, we have generated problems with high, low or even numbers for $N_C$, $N_S$ and $N_V$ relative to one another to extract the differences that arise from various combinations. We performed all of the tests of this chapter on a single core of an AMD Opteron(tm) Processor 6282 SE, on a UNIX OS, a 32GB virtual memory limit, and a 168 hour (i.e., 7 day) processing time limit.
For both exact and heuristic solutions, the transformation method outperforms the IP solution by a wide margin. The transformation method not only results in shorter computational time but also can provide accurate solutions to much larger problems. We observe that the heuristic solution of LKH is capable of obtaining the optimal solution with remarkable frequency in a very short time in comparison to the exact methods. Test instances of Tables 5.4–5.5 also shed light on factors

<table>
<thead>
<tr>
<th>Problem</th>
<th>Model</th>
<th>Exact Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Opt. val</td>
<td>Solver</td>
</tr>
<tr>
<td>rnd2-2-2n</td>
<td>8</td>
<td>920.93</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rnd3-3-3n</td>
<td>27</td>
<td>2051.85</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rnd3-3-7n</td>
<td>63</td>
<td>858.99</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rnd3-7-3n</td>
<td>63</td>
<td>3276.09</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rnd7-3-3n</td>
<td>63</td>
<td>3895.23</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rnd4-4-4n</td>
<td>64</td>
<td>2504.53</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rnd5-5-5n</td>
<td>125</td>
<td>3169.19</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rnd3-3-14n</td>
<td>126</td>
<td>570.85</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rnd3-14-3n</td>
<td>126</td>
<td>——*</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rnd14-3-3n</td>
<td>126</td>
<td>5892.84</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rnd6-6-6n</td>
<td>216</td>
<td>——</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rnd3-3-24n</td>
<td>216</td>
<td>589.32</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rnd3-24-3n</td>
<td>216</td>
<td>——</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rnd24-3-3n</td>
<td>216</td>
<td>11589.96</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rnd3-3-38n</td>
<td>342</td>
<td>475.88</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rnd3-38-3n</td>
<td>342</td>
<td>——</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rnd38-3-3n</td>
<td>342</td>
<td>17710.13</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.4: Comparative performance of the IP versus the TSP formulation of CGTSP for exact solutions. Empty entries of the table imply that an optimal tour was not verified for the instance.
Table 5.5: Comparative performance of the IP versus TSP formulation of CGTSP for heuristic solutions. NN diff column reports the percentage by which the solution of the nearest neighbor algorithm is larger than the heuristic solution.

<table>
<thead>
<tr>
<th>Problem Model</th>
<th>Opt. Solution</th>
<th>Heuristic Solution</th>
<th>Time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Size</td>
<td>Opt. val</td>
<td>Opt. gap(%)</td>
</tr>
<tr>
<td>rnd2-2-2n</td>
<td>8</td>
<td>920.93</td>
<td>14.55 %</td>
</tr>
<tr>
<td>rnd3-3-3n</td>
<td>27</td>
<td>2051.85</td>
<td>25.28 %</td>
</tr>
<tr>
<td>rnd3-3-7n</td>
<td>63</td>
<td>858.99</td>
<td>145.32 %</td>
</tr>
<tr>
<td>rnd3-7-3n</td>
<td>63</td>
<td>3276.09</td>
<td>61.54 %</td>
</tr>
<tr>
<td>rnd7-3-3n</td>
<td>63</td>
<td>3895.23</td>
<td>10.76 %</td>
</tr>
<tr>
<td>rnd4-4-4n</td>
<td>64</td>
<td>2504.53</td>
<td>38.80 %</td>
</tr>
<tr>
<td>rnd5-5-5n</td>
<td>125</td>
<td>3169.19</td>
<td>52.99 %</td>
</tr>
<tr>
<td>rnd3-3-14n</td>
<td>126</td>
<td>570.85</td>
<td>134.78 %</td>
</tr>
<tr>
<td>rnd3-14-3n</td>
<td>126</td>
<td>——*</td>
<td>51.90 %</td>
</tr>
<tr>
<td>rnd14-3-3n</td>
<td>126</td>
<td>5892.84</td>
<td>29.22 %</td>
</tr>
<tr>
<td>rnd6-6-6n</td>
<td>216</td>
<td>——</td>
<td>47.20 %</td>
</tr>
<tr>
<td>rnd3-3-24n</td>
<td>216</td>
<td>589.32</td>
<td>45.78 %</td>
</tr>
<tr>
<td>rnd3-24-3n</td>
<td>216</td>
<td>——</td>
<td>52.96 %</td>
</tr>
<tr>
<td>rnd24-3-3n</td>
<td>216</td>
<td>11589.96</td>
<td>26.47 %</td>
</tr>
<tr>
<td>rnd3-3-38n</td>
<td>342</td>
<td>475.88</td>
<td>70.34 %</td>
</tr>
<tr>
<td>rnd3-38-3n</td>
<td>342</td>
<td>——</td>
<td>52.96 %</td>
</tr>
<tr>
<td>rnd38-3-3n</td>
<td>342</td>
<td>17710.13</td>
<td>55.53 %</td>
</tr>
<tr>
<td>rnd7-7-7n</td>
<td>343</td>
<td>——</td>
<td>55.47 %</td>
</tr>
<tr>
<td>rnd8-8-8n</td>
<td>512</td>
<td>——</td>
<td>48.26 %</td>
</tr>
<tr>
<td>rnd3-3-57n</td>
<td>513</td>
<td>——</td>
<td>126.42 %</td>
</tr>
<tr>
<td>rnd3-57-3n</td>
<td>513</td>
<td>——</td>
<td>40.88 %</td>
</tr>
<tr>
<td>rnd57-3-3n</td>
<td>513</td>
<td>——</td>
<td>38.64 %</td>
</tr>
</tbody>
</table>

* Some entries of the table are empty because the value of the optimal tour is unknown.
** For all instances of the table MAX_CANDIDATES is 25 for LKH.
that may affect the computational difficulty of a problem. It appears that, for problems of similar size, the higher the number of subclusters, the harder it is to solve the instance.

The test instances of Table 5.6 were designed to push the computational limitations of the TSP transformation on large scale test instances. Following the results of the previous table which demonstrated that LKH is capable of obtaining high quality solutions in relatively short time, we push the algorithm to its limits on the CGTSP instances. For large instances, the trade-off between solution time and solution quality is crucial. In our experiments with LKH, we observed that the parameter with the highest impact on the solution quality is MAXIMUM_CANDIDATES. So, we report the results for two polar values 25 and 50 for the relevant parameter. The starting arrangement of nodes for LKH does not necessarily correspond to a valid CGTSP solution, therefore, LKH may or may not output even a feasible CGTSP solution at all.

**Table 5.6: Performance of LKH on large scale instances for MAX_CANDIDATES = 25 and 50. NN diff column reports the percentage by which the solution of the nearest neighbor algorithm is larger than the heuristic solution.**

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Max Cand.</th>
<th>Feasible</th>
<th>Solution</th>
<th>NN diff</th>
<th>Time(hh:mm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>rnd10-10-10n</td>
<td>1000</td>
<td>25</td>
<td>YES</td>
<td>6444.41</td>
<td>45.16%</td>
<td>01:36</td>
</tr>
<tr>
<td></td>
<td></td>
<td>50</td>
<td>YES</td>
<td>6511.85</td>
<td>43.66%</td>
<td>10:35</td>
</tr>
<tr>
<td>rnd12-12-12n</td>
<td>1728</td>
<td>25</td>
<td>YES</td>
<td>8773.07</td>
<td>49.13%</td>
<td>02:28</td>
</tr>
<tr>
<td></td>
<td></td>
<td>50</td>
<td>YES</td>
<td>8800.82</td>
<td>48.66%</td>
<td>10:57</td>
</tr>
<tr>
<td>rnd14-14-14n</td>
<td>2744</td>
<td>25</td>
<td>YES</td>
<td>11111.68</td>
<td>48.15%</td>
<td>03:35</td>
</tr>
<tr>
<td></td>
<td></td>
<td>50</td>
<td>YES</td>
<td>10594.62</td>
<td>55.38%</td>
<td>14:17</td>
</tr>
<tr>
<td>rnd16-16-16n</td>
<td>4096</td>
<td>25</td>
<td>YES</td>
<td>14400.34</td>
<td>16.59%</td>
<td>05:43</td>
</tr>
<tr>
<td></td>
<td></td>
<td>50</td>
<td>YES</td>
<td>12068.19</td>
<td>39.12%</td>
<td>24:09</td>
</tr>
<tr>
<td>rnd18-18-18n</td>
<td>5832</td>
<td>25</td>
<td>YES</td>
<td>16081.87</td>
<td>27.93%</td>
<td>08:43</td>
</tr>
<tr>
<td></td>
<td></td>
<td>50</td>
<td>YES</td>
<td>13669.79</td>
<td>50.51%</td>
<td>28:35</td>
</tr>
<tr>
<td>rnd20-20-20n</td>
<td>8000</td>
<td>25</td>
<td>NO</td>
<td>NA</td>
<td>NA</td>
<td>15:29</td>
</tr>
<tr>
<td></td>
<td></td>
<td>50</td>
<td>YES</td>
<td>16643.49</td>
<td>36.36%</td>
<td>44:08</td>
</tr>
<tr>
<td>rnd22-22-22n</td>
<td>10648</td>
<td>25</td>
<td>NO</td>
<td>NA</td>
<td>NA</td>
<td>31:49</td>
</tr>
<tr>
<td></td>
<td></td>
<td>50</td>
<td>NO</td>
<td>NA</td>
<td>NA%</td>
<td>72:35</td>
</tr>
</tbody>
</table>

The largest example where LKH found a reasonable solution has 8,000 nodes. It should be emphasized that that the NN diff values in Table 5.6 are relatively consistent with Table 5.5 for MAX_CANDIDATES=50, indicating that the TSP trans-
formation is effective on large instances where exact solutions are computationally impractical. We can also observe that for larger values of MAX_CANDIDATES, the solution quality diverges significantly.

5.4 CGTSP and Logistics Modeling

The power and range of applications of TSP in logistics have been demonstrated in the literature, for example in modern transportation systems (e.g., hybrid electric vehicle transportation [36] and smart city logistics networks [128]). As an extension of TSP, a CGTSP model may incorporate a more diverse variety of complexities and situations compared to a classical TSP model. For the first time, our transformation in this chapter enables for a computationally viable method to attempt to solve CGTSP, and hence for its real world applications to be tackled. To demonstrate the power and flexibility of a CGTSP model, we present two potential applications of CGTSP in logistics, namely, warehouse logistics, and delivery systems that are both important in the field of logistics management. Note that warehouse logistics and delivery systems are also called inbound distribution and outbound distribution [24], respectively. A computational study of these problems is also included here.

5.4.1 Warehouse Logistics (Robotic Automated Storage and Retrieval System)

A conventional robotic automated storage and retrieval system (ASRS) consists of an overhead stacker crane, an input/output (I/O) conveyor, and multiple racks for storing and retrieving items. Efficient operation of an ASRS relies on the crane scheduling, which decides on the sequence of crane moves for processing specific storage and retrieval requests. The crane is given lists of items from different
locations in the warehouse and delivers them to the I/O conveyor.

The survey papers by [22] and [79] summarize the recent work on traditional ASRSs. Despite a wide range of studies on the topic, there is a shortage of attention on the impact of modern technologies such as robotics on warehousing systems [21]. Many industries commonly use robots as a means of automation for material handling [52]. For this reason, the need for logistics models that help us with understanding and managing the impact of robotics on productivity is evident. For example, in a warehouse that uses robots for collecting items, logistics model may not only be used for efficient movement decision of robots in the warehouse, but also may be employed for arranging items in a warehouse to maximize efficiency.

In the context of automated item collection in a warehouse, a TSP model is a natural starting point; a robot begins at a conveyor, it then must travel to collect each item, and finally returns to the conveyor. However, certain common complexities are difficult to incorporate in a classical TSP model. For example, the underlying assumption is that each type of item is available only in one location, not more. Therefore, sequencing the requests of a crane can be modeled as a TSP for each received order, as detailed in [118]. However, storing the same type of items in multiple locations sometimes helps with the efficiency of a warehouse, and this is very common in practice. For instance, technologies such as RFID-based wireless identification technologies allow for item localization. More precisely, an RFID tag is mounted on each item that needs to be monitored and then an RFID reader is mounted on the robot in this type of ASRS [92, 86]. This enables quick detection of multiple storage locations of a particular item in the warehouse. In such situations, a model with more flexibility than a TSP model, such as a CGTSP model, would be more suitable.

In a CGTSP model of ASRS, each item in an order corresponds to a particular
subcluster, and different nodes within a subcluster correspond to the different locations where the item is stored. The robot then only needs to choose only one of the locations for collecting the item. Furthermore, a cluster represents an order, i.e., a list of items. Often, an order must be stacked together by the robot before the delivery to I/O conveyor. If the robot is required to collect multiple orders, where items of one order are to be all collected before moving on to another order, different orders are represented as different clusters in CGTSP. Note that if two different orders contain the same item, the item in the two orders must each be separately represented by a distinct subcluster within the corresponding clusters.

A CGTSP model of ASRS was first introduced in [43], and the primary solution method was the cross-entropy (CE) metaheuristic, a meta-heuristic approach. The reason for choosing a meta-heuristic was the difficulty of solving CGTSP instances optimally by exact methods. As demonstrated in Tables 5.4–5.6, the CE meta-heuristic is strongly dominated by the TSP transformation method. As a consequence, the power of a CGTSP in being applied to ASRS is also significantly enhanced by the use of the TSP transformation method.

One of the potential applications of CGTSP in ASRS is to inform the arrangement of items in a warehouse. For instance, we have two alternatives for the storage policy: random storage and volume-based storage policies [118]. Figure 5.4 illustrates examples of such arrangements. Different colors represent different types of items. Since a particular type of item is stored at multiple locations, there is a set of shelves in different racks with a corresponding color. The robot visits one of these shelves for a particular item in an order.

It is typical that some items attract more demand compared to other items. For example, 80% of sales may come from only 20% of the total items, i.e., the high demand items. A CGTSP model may explore where these high-demand items are
I/O

(a) Robotic ASRSs with random storage policy.

Highly demanded items

(b) Robotic ASRSs with volume-based storage policy.

Figure 5.4: The rectangular layout of a robotic ASRS with random and volume-based storage policies.

stored in the warehouse. Item locations of Figure 5.4(a), are randomly assigned. In contrast, in Figure 5.4(b), the locations close to the I/O conveyor are predominantly assigned to high-demand items in order to implement a volume-based storage policy.

In the Table 5.7, we have included a simple example of comparisons between two warehouse arrangements. Consider a warehouse where the stored items are in the two categories of high-demand and low-demand items, consisting of 20% and 80% of total items respectively. We randomly generate orders for which we presume that there is a 80% chance that a randomly selected item in an order is marked as a high-demand item. Furthermore, there are 400 possible storage locations in the warehouse, and each item may be stored at multiple locations. The ASRS test instances are labeled as “ASRS_i<A>r<L>_I<<type>>", where A is the total number of available items in the warehouse, L is the number of locations where
each item is stored, and $I$ is the total number of items in all orders. It is clear that the size of the problem is $n = L \times I$. The type of the instance determined whether the locations of all items is randomly selected in the warehouse ($type = \text{rnd}$), or if there is at least one location for all high-demand items near the I/O conveyor ($type = \text{alt}$). These instances are solved by LKH with MAX_CANDIDATES = 50 for the sake of simplicity of the comparison.

The table demonstrates that the random arrangement is, perhaps surprisingly, superior in efficiency relative to the alternative, which is volume-based storage policy. Even though the movement of the robot between high-demand items near the I/O conveyor is less time-consuming in the alternative arrangement, moving between any other pairs of locations is likely to be more time-consuming. In our generated test instances, the losses outweigh the benefits. We note that the results of our experiment should not be regarded as a general principle for the efficiency of the warehouse, as they rely heavily on the assumptions. Rather, our emphasis is on demonstrating the comparative power of CGTSP modeling in comparing varieties of warehouse arrangements.

5.4.2 Delivery Logistics (Drone-assisted Parcel Delivery Service)

The rise of giant e-commerce companies like Amazon, JD.com, and Alibaba has had a great impact on the development of drone delivery technology. There are many ways of incorporating drones into delivery systems. One way is that a delivery truck carries drones that assist the truck in parcel delivery tasks. While the delivery truck moves from one customer to the next customer, drones pick up parcels from the truck and serve more close-by customers. Each drone returns to the truck after the corresponding delivery to pick up another parcel, which is usually much closer than the depot. This cooperation between a truck and drones not only significantly reduces the travel time of a truck, but it also addresses the
Table 5.7: Comparison of random and alternative warehouse arrangements. The \( \text{rnd imprv.} \) column reports the percentage by which the arrangement improves the random storage arrangement.

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Solution</th>
<th>( \text{rnd imprv.}(%) )</th>
<th>Time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASRS_i80r4_16rnd</td>
<td>64</td>
<td>2905.52</td>
<td>0%</td>
<td>30</td>
</tr>
<tr>
<td>ASRS_i80r4_16alt</td>
<td></td>
<td>4330.78</td>
<td>-49.05%</td>
<td>19</td>
</tr>
<tr>
<td>ASRS_i125r5_16rnd</td>
<td>80</td>
<td>5621.31</td>
<td>0%</td>
<td>33</td>
</tr>
<tr>
<td>ASRS_i125r5_16alt</td>
<td></td>
<td>6330.00</td>
<td>-12.60%</td>
<td>40</td>
</tr>
<tr>
<td>ASRS_i80r4_64rnd</td>
<td>256</td>
<td>9499.19</td>
<td>0%</td>
<td>253</td>
</tr>
<tr>
<td>ASRS_i80r4_64alt</td>
<td></td>
<td>12694.68</td>
<td>-33.64%</td>
<td>237</td>
</tr>
<tr>
<td>ASRS_i125r5_64rnd</td>
<td>320</td>
<td>12521.79</td>
<td>0%</td>
<td>446</td>
</tr>
<tr>
<td>ASRS_i125r5_64alt</td>
<td></td>
<td>16114.82</td>
<td>-28.69%</td>
<td>497</td>
</tr>
<tr>
<td>ASRS_i80r4_256rnd</td>
<td>1024</td>
<td>38397.53</td>
<td>0%</td>
<td>2514</td>
</tr>
<tr>
<td>ASRS_i80r4_256alt</td>
<td></td>
<td>47768.26</td>
<td>-24.40%</td>
<td>3524</td>
</tr>
<tr>
<td>ASRS_i125r5_256rnd</td>
<td>1280</td>
<td>27659.16</td>
<td>0%</td>
<td>2289</td>
</tr>
<tr>
<td>ASRS_i125r5_256alt</td>
<td></td>
<td>35914.11</td>
<td>-29.84%</td>
<td>3048</td>
</tr>
<tr>
<td>ASRS_i80r4_1024rnd</td>
<td>4096</td>
<td>103263.79</td>
<td>0%</td>
<td>27680</td>
</tr>
<tr>
<td>ASRS_i80r4_1024alt</td>
<td></td>
<td>138174.98</td>
<td>-33.81%</td>
<td>59170</td>
</tr>
<tr>
<td>ASRS_i125r5_1024rnd</td>
<td>5120</td>
<td>118131.25</td>
<td>0%</td>
<td>49948</td>
</tr>
<tr>
<td>ASRS_i125r5_1024alt</td>
<td></td>
<td>138174.16</td>
<td>-16.97%</td>
<td>55511</td>
</tr>
</tbody>
</table>

A major research effort has started to concentrate on this mode of delivery over the last couple of years. Currently, there is no comprehensive survey paper on drone-assisted PDS. However, for some individual studies on drone-assisted PDS, see [95], [3], and [120]. These studies illustrate that TSP is a common modeling approach to tackle variations of drone-assisted PDS when the PDS employs a single drone. However, a significant weakness of the TSP model is that it is not easy to incorporate the use of multiple drones into the model. Recently, researchers have begun to study methods of bundling customers in drone-assisted PDS [95, 59]. In these studies, it is assumed that the truck does not collaborate with drones. In other words, the truck serves multiple customers on a delivery tour, while the drones are constrained to perform back and forth trips to the depot.
A CGTSP model of drone-assisted PDS allows for adaptations for various common situations. We divide the customer locations into clusters and subclusters, as depicted in Figure 5.5. A truck visits only a single node in each subcluster, and the drones visit all other nodes in the subcluster. Clusters also allow us to incorporate common situations where the preference is to visit all customers located in a region before moving to another region. For example, zip code constraints for delivery [4], governmental regulations of drones [102], and toll charges of the truck [121] may constitute situations where clustering is beneficial or even necessary. Even when such constraints do not exist, we may use clustering algorithms that identify nearby vertices may be used to identify the clusters (e.g., in [41]), and later, the number of available drones would determine the further division of each cluster into subclusters. It is noteworthy that the drone-assisted PDS considered in [95, 59] is a special case of a CGTSP where $|S_{0,1}| > 1$ and $|S_{c,s}| = 1$ if $c \neq 1$ and $s \neq 1$.

Consider the movement of the truck between two customers from locations $u$ to $v$. In the CGTSP model, the delivery points are represented by nodes of different subclusters, say $u \in S_{c,s}$ and $v \notin S_{c,s}$. The movement of the truck from $u$ to $v$ implies the following activities take place:

**Delivery by truck:** The truck moves from $v$ toward the location of customer $v$, where $u \in S_{c,s}$ and $v \notin S_{c,s}$, to deliver a parcel to $v$.

*Figure 5.5:* An example of a CGTSP route for a drone-assisted PDS in Melbourne central business district.
**Delivery by drones:** Meanwhile, each drone fly carries its assigned parcel from the truck at $u$ and deliver the parcel to the corresponding customer $p \in S_{c,s} \setminus \{u\}$.

**Return of the drones to the truck:** After each drone delivers the parcel to its delivery point, the empty drone move towards $v$ to meet with the truck and load the next parcel.

For determining the cost of a CGTSP edge $(u,v)$, the cost associated with each of the above activities may be defined separately. For example, let us suppose the objective is to minimize delivery times, and the travel time associated with the travel from $u$ to $v$ for the truck, from $u$ to $p$ for a loaded drone, and from $p$ to $v$ for an empty drone are denoted $d^f(u,v)$, $d^{ld}(u,p)$ and $d^{ed}(p,v)$, respectively. Then the time of travel associated with edge $(u,v)$ is

\[ \omega(u,v) = \max_{p \in S_{c,s} \setminus \{u\}} \{d^f(u,v), d^{ld}(u,p) + d^{ed}(p,v)\} \]  

(5.3)

Note that $\omega(u,v)$ is only defined if the subclusters of $u$ and $v$ are not the same, i.e., $u \in S_{c,s}$ and $v \notin S_{c,s}$, while $u$ and $p$ belong to the same subcluster, i.e., $u, p \in S_{c,s}$.

As mentioned earlier, the reason why we model variants of ASRS and PDS is to demonstrate the power and range of applications of CGTSP in logistics inside and outside the warehouse. Moreover, the results obtained in this section show that CGTSP can be applied for problems with different structures. For a CGTSP, we may visit only a set of nodes, such as in a robotic ASRS; or it may be necessary to visit all nodes, such as the drone-assisted PDS. This is due to the fact that the truck plays the role of the salesman in CGTSP, rather than the drones. Thus, we should find a way to integrate drones’ travel times inside the truck travel time. For this purpose, it is enough to compute all entries of the CGTSP length matrix.
beforehand, which is a trivial task and is described in this section. Clearly, the CGTSP structure offers the possibility of creating nuanced modeling of different problems, with the additional advantage that the transformation to a classical TSP described in this chapter enables the use of powerful TSP solvers for efficient solution of the CGTSP problem.

To analyze the performance of this approach, we have used GTSPLIB instances\(^1\) to generate CGTSP instances corresponding to the drone-assisted PDS and compare the impact of the different number of drones in Table 5.8. In the GTSPLIB library the nodes of the selected TSPLIB test instances\(^2\) are divided into regions (clusters of close-by nodes) by a simple procedure that is described in [41].

If the size of any particular cluster is larger than the number of drones plus one, we further subdivide the cluster into subclusters where the truck only visits one of the nodes from each subcluster, and the drones visit the rest of the nodes. We assume that loaded drones and empty drones are 50% and 100% faster than the truck respectively. We also presume that one unit of distance traveled by truck corresponds to one unit of time. As a measure of improvements made by adding the drones, we compare the time values to the case where the drones are not employed. It is interesting to note that if we employ zero drones, the problem is equivalent to a CTSP. Furthermore, if there are enough drones in each step to cover an entire cluster, then the problem reduces to a GTSP.

Perhaps the most significant observation in Table 5.8 is that Concorde was able to verify the optimality of the tour for most of these instances. Only the values marked by a star sign do not correspond to optimal values. Even in those instances, Concorde failed due to implementation choices\(^2\) of Concorde rather than

---

\(^1\)http://www.cs.rhul.ac.uk/home/zvero/GTSPLIB/

\(^2\)In these tests, Concorde encountered the “BIG GUY” error, which indicates that the algorithm is encountering numbers that are require more memory than the allowed by the choice of container. We expect this issue to be easily overcome in an improved implementation of the same algorithmic method.
Table 5.8: Drone-assisted PDS on GTSPLIB instances with different number of drones. The percentage of improvement compared to using no drones is reported in the no drones imprv column. The numbers marked by a * were obtained by a heuristic algorithm.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Size</th>
<th>Clusters</th>
<th>Drones</th>
<th>Solution</th>
<th>No drones imprv. (%)</th>
<th>Time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4ulysses16</td>
<td>16</td>
<td>25</td>
<td>0</td>
<td>74.20</td>
<td>0%</td>
<td>3</td>
</tr>
<tr>
<td>max</td>
<td></td>
<td></td>
<td></td>
<td>54.57</td>
<td>26.45%</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td>60.57</td>
<td>18.26%</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td>56.23</td>
<td>24.22%</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td>54.57</td>
<td>26.45%</td>
<td>4</td>
</tr>
<tr>
<td>11eil51</td>
<td>51</td>
<td>25</td>
<td>0</td>
<td>45183.15</td>
<td>0%</td>
<td>3</td>
</tr>
<tr>
<td>max</td>
<td></td>
<td></td>
<td></td>
<td>22169.01</td>
<td>50.94%</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td>30709.82</td>
<td>32.030%</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td>25559.12</td>
<td>43.43%</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td>22169.01</td>
<td>50.94%</td>
<td>6</td>
</tr>
<tr>
<td>25pr124</td>
<td>124</td>
<td>25</td>
<td>0</td>
<td>565817.48</td>
<td>0%</td>
<td>306</td>
</tr>
<tr>
<td>max</td>
<td></td>
<td></td>
<td></td>
<td>266366.71</td>
<td>52.92%</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td>434980.16</td>
<td>23.12%</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td>312411.42</td>
<td>44.79%</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td>291240.12</td>
<td>48.53%</td>
<td>6</td>
</tr>
<tr>
<td>40krob200</td>
<td>200</td>
<td>40</td>
<td>0</td>
<td>1018845.89</td>
<td>0%</td>
<td>510</td>
</tr>
<tr>
<td>max</td>
<td></td>
<td></td>
<td></td>
<td>468862.62</td>
<td>53.98%</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td>766236.70</td>
<td>24.79%</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td>576493.73</td>
<td>43.42%</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td>509406.36</td>
<td>50.00%</td>
<td>8</td>
</tr>
<tr>
<td>87gr431</td>
<td>431</td>
<td>87</td>
<td>0</td>
<td>252039.05*</td>
<td>0%</td>
<td>2852</td>
</tr>
<tr>
<td>max</td>
<td></td>
<td></td>
<td></td>
<td>102854.33</td>
<td>59.19%</td>
<td>2350</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td>230401.47</td>
<td>8.58%</td>
<td>952</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td>195464.92</td>
<td>22.47%</td>
<td>2901</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td>173116.45*</td>
<td>31.31%</td>
<td>4318*</td>
</tr>
<tr>
<td>157rat783</td>
<td>783</td>
<td>157</td>
<td>0</td>
<td>2166920.50</td>
<td>0%</td>
<td>1113</td>
</tr>
<tr>
<td>max</td>
<td></td>
<td></td>
<td></td>
<td>1008301.23</td>
<td>53.46%</td>
<td>911</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td>1581884.37</td>
<td>27.00%</td>
<td>4170</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td>1232516.12</td>
<td>43.12%</td>
<td>2527</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td>1232849.71*</td>
<td>43.11%</td>
<td>972*</td>
</tr>
<tr>
<td>217vm1084</td>
<td>1084</td>
<td>217</td>
<td>0</td>
<td>4241722.50*</td>
<td>0%</td>
<td>6158*</td>
</tr>
<tr>
<td>max</td>
<td></td>
<td></td>
<td></td>
<td>1847174.49</td>
<td>56.43%</td>
<td>1609</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td>3289275.74*</td>
<td>22.45%</td>
<td>6846*</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td>2564049.17*</td>
<td>39.55%</td>
<td>2265*</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td>2273911.94*</td>
<td>46.39%</td>
<td>8965*</td>
</tr>
</tbody>
</table>

the limitation of the algorithmic method. Since the problem sizes are reasonable comparing to real situations (the number of customers that a delivery truck might be expected to serve), this is a highly promising result for the application of CGTSP to drone-assisted PDS.
Table 5.8 also illustrates that the number of drones has a significant impact on the time efficiency of the delivery. As an integrated design and sequencing problem, using additional drones may come with some costs to a delivery company. Although it is outside the scope of this study, a CGTSP model may be utilized by companies for balancing the reduction of delivery times with the cost of operating additional drones. Therefore, results can accomplish the economic benefits and provide a set of managerial insights for e-commerce companies to determine whether it is technically feasible and profitable to apply drones for delivery purposes.
Chapter 6

Conclusion and Future Work

In this work, I have investigated three variations of the Traveling Salesman Problem (TSP), namely, the Hamiltonian Cycle Problem (HCP), the Sparse Traveling Salesman Problem (STSP), and the Clustered Generalized Traveling Salesman Problem (CGTSP). For each of these problems, competitive algorithmic solutions are developed and their performance is extensively documented. I have also formulated mathematical models based on the variants of TSP for solving applied problems in logistics and bioinformatics.

In Chapter 2, I have discussed, studied and expanded on the algorithmic principles of the Snake and Ladders Heuristic (SLH) for solving the HCP. The dominant tour improvement heuristic method based on the Lin-Kernighan heuristic is to search for transformations that iteratively improve the value of the tour. Instead, we have proposed a new perspective on the concept of improvement and introduced the idea of a Blind Improvement Space (BIS), a mapping from a tour to a set of transformations where at least one of the transformation is guaranteed to improve the tour. The BIS-based transformations operate fundamentally different from the previous methods for heuristic solutions. Nevertheless, they are demonstrated to be a highly effective approach, especially in circumstances where the long-established tour-improvement techniques fall short. The performance of the new
algorithmic method is on par with the state-of-the-art algorithms of TSP, such as Concorde and LKH, on instances of HCP. While the BIS-based transformations have, to date, been only applied to HCP, the theoretical work of this thesis creates a potential path for generalizing the concepts for a new algorithm for solving the general case of TSP. Fundamentally, the new concept of improvement and the concept of the BIS are intelligible for other combinatorial optimization problems. A potential topic for future is to define BIS transformations for other NP-complete problems, such as SAT, and investigate whether there is any potential for creating new heuristic algorithms for these problems.

In Chapter 3, an extensive computational study was conducted on the state-of-the-art algorithms for HCP and TSP on difficult instances of the HCP. Our approach has been to find difficult instances of HCP, and assume that these will also provide difficult benchmark instances for TSP, or at least stress TSP algorithms in ways that the existing benchmark sets do not. This was because the structure of HCP allowed us to generate examples that are structurally different from examples in these existing TSP benchmark sets. We have demonstrated that studying small but challenging outlier examples can provide an insight into the weaknesses of an algorithm. Such insight could lead to finding areas of improvement for the algorithmic approaches, as well as helping to differentiate between the algorithms.

A future endeavour is to theoretically extract the features of the difficult instances. For example, we have witnessed many instances where a high level of symmetry is combined with low number of Hamiltonian cycles, often results in difficult instances for various approaches. However, the type of structural symmetry that contributes to difficulty is not theoretically understood, and other instances with seemingly similar symmetrical properties were trivial to solve. Studying what makes these problems difficult could not only provide us information on how to construct better benchmark examples, it can also shed light on the nature of some difficulties of solving the TSP and the HCP in general. Furthermore, one could
look at different problems and study small but difficult instances in other optimisation settings.

In Chapter 4, effective algorithmic approaches for solving the Sparse Traveling Salesman Problem were analyzed. Effective algorithms from the literature were combined with new approaches which results in a new hybrid method for solving STSP. The STSP formulation was then applied for solving two problems. The first application was the Time-Dependant Traveling Salesman Problem (TDTSP), which is a well-known variation of the TSP. I have developed an efficient transformation method from TDTSP to STSP which enables the well-developed TSP solvers to be taken advantage of when solving TDTSP. The second application was an STSP formulation for DNA-assembly, where I introduced a TSP overlap graph for the de novo assembly. The conventional belief in the Bioinformatics community is that a TSP model is not computationally practical for the real-world DNA-assembly problems. I have designed a test, by simulating the structure of the TSP overlap graphs, to challenge this convention. These tests indicate that a suitably well-designed TSP model, paired with a clever TSP algorithm, could potentially be useful for the state-of-the-art DNA sequencing technology.

There are three primary potential research topics that arise from Chapter 4. First, since the implemented versions of the STSP algorithms in this thesis were developed as proof of concept, there is potential to significantly improve their performance. Second, the special structural characteristics of TDTSP instances should be applied for solving the STSP formulation of TDTSP. Being a well-studied variation of TSP, the structural properties of TDTSP are well studied in the literature. In particular, some or all of the TDTSP constraints developed by Abeledo et al. [1] may be added to Concorde, or another TSP solver, and will almost certainly improve the performance on the STSP formulation of TDTSP instances significantly. Finally, the initial results of the chapter point to the potential for a DNA assembly solver that relies on the STSP formulation. This would involve implementation
of the appropriate error correction and overlap calculations for the STSP formulation. Since the STSP formulation of DNA-assembly may arguably avoid many potential mistakes made by other approaches, such a solver offers potential for the third generation sequencing technology where the total number of reads could be a few million or smaller.

In Chapter 5, we have considered an extended variant of the Traveling Salesman Problem (TSP), called the Clustered Generalized Traveling Salesman Problem (CGTSP), motivated by its many potential applications in logistics. We have developed a transformation that first converts the CGTSP to a constrained TSP, which was then transformed to a classical TSP. This transformation is effective in the sense that the size of the resultant TSP instance is equal to the size of the CGTSP instance, and enables practically solutions to these instances. The CGTSP provides a higher level of flexibility for modeling compared to the classical TSP, because it is an extension of the TSP. We have illustrated two applications in modern logistics where the structure of CGTSP may be successfully exploited, to highlight the broad and practical applicability of a CGTSP model.

We have compared this approach both in the context of exact and heuristic to a recent IP formulation of CGTSP [43]. The exact solutions of Concorde TSP solver were compared to Cplex, and the solutions of Helsgaun’s Lin Kernighan Heuristic (LKH) are compared to a cross-entropy (CE) metaheuristic algorithm. The experiments demonstrated a staggering dominance of the transformation technique compared to the IP formulation to the point that an exact solution via the transformation technique was usually obtained faster than a heuristic solution by the alternative method. We have also examined the size of the problems that can be modeled by a CGTSP model by obtaining reasonable heuristic solutions by LKH, for randomly generated problems of up to 10,000 nodes.

Regarding the applications of the CGTSP to modern logistics, we have considered
one application inside a warehouse, namely Automated Storage/Retrieval Systems, and another problem outside of the warehouse, namely the drone-assisted parcel delivery systems (PDS). For the former problem, we have compared two storage policies in a warehouse, in particular, a random policy and a volume based policy. For the latter problem, we have examined the impact of the number of drones on the delivery time of drone-assisted PDS for problems of a third party benchmark set (GTSPLIB benchmark library). It is worth noting that for the drone-assisted PDS, the CGTSP model was especially promising as Concorde was able to verify an optimal solution for most of the CGTSP problems.

The work of Chapter 5 sets out a road-map for a set of potential future research topics on CGTSP. First, similar transformation techniques may be developed for other cluster-based variations of the TSP. Second, the constrained TSP formulation may offer a superior alternative to the classical TSP formulation depending on the choice of the TSP solver. Adding the constraints to solvers like Concorde which, in principle, can incorporate constraints may lead to faster and higher quality solutions. Second, fine-tuning TSP heuristics for the structure of the transformed problem will most certainly lead to higher quality solutions. For example, LKH relies heavily on sparsification and ranking the edges and it currently is not informed about the special structure of CGTSP instances. We anticipate that fine-tuning the candidate selection in LKH will lead to considerable improvements in the quality of the solutions. Third, for the drone-assisted PDS, further research is required for algorithms that separate the delivery locations into better delivery regions (nodes clustering). Finally, since we have only briefly discussed considered two applications of CGTSP, there is an opportunity to conduct a more comprehensive study for each application and also investigate other applications of CGTSP.
Appendix A

A.1 Pilot Computational Study on Random TDTSP Instance

In this Appendix, we conduct an experiment on randomly generated instances of TDTSP. These instances are solved by first converting them to TSP as described in Section 4.3, and then submitting them to various TSP solver. The TDTSP instances are generated as follows. For each \( t = 1, 2, \ldots n \), a symmetric matrix of size \( n \) is generated that contains random weights of up to 100 and the \( i,j \) entries are the \( d(t,i,t) \) distances of TDTSP. The instances are then converted to a TSP by the conversion described in Section 4.3. Then, the asymmetric TSP instances are transformed to symmetric TSP instances by a simple conversion that doubles the sizes of the graph. We performed all of the tests on a single core of an AMD Opteron(tm) Processor 6282 SE on a UNIX OS server.

The converted TDTSP instance contain special rare structures. The degree of every node is equal to the number of steps associated with the problem. Therefore, even though these instances are relatively sparse, there are still many edges compared to graphs with bounded degrees. More importantly, there are exponentially
Table A.1: Performance of TSP algorithms on the randomly generated TDTSP instances. The percentage of difference with the optimal tour is reported in the Gap column. A 0% gaps implies that the optimal tour is obtained.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Steps</th>
<th>TSP Size</th>
<th>Opt.</th>
<th>Concorde</th>
<th>Concorde-SLH</th>
<th>sLKH</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Gap</td>
<td>Time (s)</td>
<td>Gap</td>
</tr>
<tr>
<td>tdtsp-t10</td>
<td>10</td>
<td>200</td>
<td>96</td>
<td>0%</td>
<td>7.57</td>
<td>0%</td>
</tr>
<tr>
<td>tdtsp-t12</td>
<td>12</td>
<td>288</td>
<td>106</td>
<td>0%</td>
<td>14.53</td>
<td>0%</td>
</tr>
<tr>
<td>tdtsp-t14</td>
<td>14</td>
<td>392</td>
<td>114</td>
<td>0%</td>
<td>50.11</td>
<td>0%</td>
</tr>
<tr>
<td>tdtsp-t16</td>
<td>16</td>
<td>512</td>
<td>100</td>
<td>0%</td>
<td>35.0</td>
<td>0%</td>
</tr>
<tr>
<td>tdtsp-t18</td>
<td>18</td>
<td>648</td>
<td>112</td>
<td>0%</td>
<td>113.58</td>
<td>0%</td>
</tr>
<tr>
<td>tdtsp-t20</td>
<td>20</td>
<td>800</td>
<td>110</td>
<td>0%</td>
<td>203.77</td>
<td>0%</td>
</tr>
<tr>
<td>tdtsp-t22</td>
<td>22</td>
<td>968</td>
<td>104</td>
<td>0%</td>
<td>1544.81</td>
<td>0%</td>
</tr>
<tr>
<td>tdtsp-t24</td>
<td>24</td>
<td>1152</td>
<td>116</td>
<td>0%</td>
<td>1332.47</td>
<td>0%</td>
</tr>
<tr>
<td>tdtsp-t26</td>
<td>26</td>
<td>1352</td>
<td>117</td>
<td>0%</td>
<td>5107.03</td>
<td>0%</td>
</tr>
<tr>
<td>tdtsp-t28</td>
<td>28</td>
<td>1568</td>
<td>125</td>
<td>0%</td>
<td>23849.09</td>
<td>0%</td>
</tr>
<tr>
<td>tdtsp-t30</td>
<td>30</td>
<td>1800</td>
<td>121</td>
<td>0%</td>
<td>45805.55</td>
<td>0%</td>
</tr>
</tbody>
</table>

many TSP tours associated with TDTSP tours, and often many edge exchanges are required to move from one TSP tour to another TSP tour. Such a structure creates many local minimum “traps” that the tour improvement algorithms find difficult to navigate.

It is not a surprise that the Concorde algorithm is the most effective method on these instances compared to the other TSP algorithms. The performance of Concorde-SLH closely follows the performance of Concorde. The slight under-performance of Concorde-SLH compared to Concorde suggests that SLH has not added to the Concorde algorithm due to the unusual structure of the converted TDTSP instances. Furthermore, the performance of sLKH is inconsistent throughout the Table. The erratic performance of sLKH appears to be due to the exponentially many local minimums associated with valid solutions to the TDTSP tours.
Bibliography


[60] Held, M., Karp, R. M. (1962). A dynamic programming approach to sequenc-


