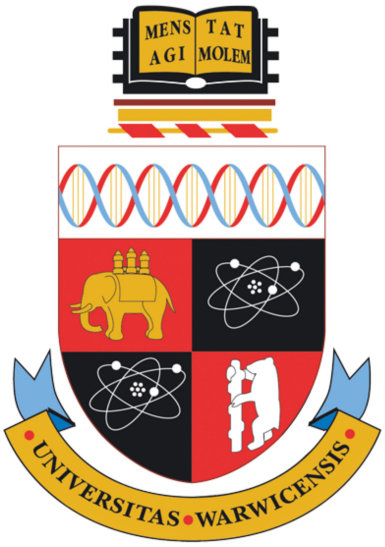


TSP and Its Variants: Use of Solvable Cases in Heuristics

by

Mengke Wang

Thesis

Submitted to the University of Warwick

for the degree ofe

Doctor of Philosophy in Business and Management

Warwick Business School

December 2022

WARWICK
THE UNIVERSITY OF WARWICK

Contents

Contents i

List of Tables iv

List of Figures vi

Acknowledgments ix

Declarations x

Abstract xi

Chapter 1 Introduction and Related Works 1

1.1 Review of Problems and Background 1

1.2 Solvability of TSP-Related Problems 5

1.3 Structure of Thesis and Contributions 9

1.4 Notation 11

Chapter 2 Kalmanson Heuristics for the TSP 12

2.1 Introduction and Related Works 12

2.2 Heuristics Motivated by Kalmanson Matrix 17

2.2.1 Classical Heuristics and Theoretical Properties for the TSP . 17

2.2.2 Kalmanson Heuristics and Theoretical Properties for the TSP 22

2.3 Empirical Investigation of Kalmanson Heuristics 27

2.4 Conclusions 37

Chapter 3 Simple Heuristics for the CTSPP 40

3.1 Introduction and Related Works 40

3.2 Problem Formulation and Simple Heuristics 43

3.2.1 Problem Formulation . . . 43
3.2.2 Simple Tour Construction Heuristics . . . 44
3.2.3 Classical Simple Tour Improvement Heuristics . . . 48
3.3 Heuristics Based on Dynamic Programming . . . 51
3.3.1 Pyramidal Heuristic . . . 51
3.3.2 Chains Heuristic . . . 55
3.3.3 Sliding Window Heuristic . . . 57
3.4 Computational experiments . . . 59
3.4.1 Experiment design . . . 59
3.4.2 Experiment Results . . . 62
3.5 Conclusions and Future Research . . . 73

Chapter 4 Heuristics for the Special CTSPP 75
4.1 Introduction and Related Works . . . 75
4.2 Extended Theory of Line-CTSPP . . . 78
4.2.1 Line-CTSPP: Classical Simple Case . . . 78
4.2.2 Line-CTSPP: Fixed Exit Point . . . 82
4.2.3 Line-CTSPP: Part CTSPP Line with a Fixed Exit . . . 84
4.2.4 Line-CTSPPW . . . 87
4.2.5 Line Heuristic . . . 89
4.3 CTSPP on SUM Matrix . . . 90
4.3.1 Theoretical Result of CTSPP on SUM Matrix . . . 90
4.3.2 Two Cases of CTSPP on SUM Matrix . . . 92
4.4 CTSPP Heuristics Based on the Convex Hull . . . 94
4.4.1 Path TSP on Convex Hull . . . 94
4.4.2 Up-Down Heuristic for CTSPP . . . 101
4.4.3 Two-Ray Heuristic for CTSPP . . . 104
4.5 Computational Experiments . . . 111
4.5.1 Experiment Design . . . 111
4.5.2 Experimental Results and Implications . . . 115
4.6 Conclusions and Future Research . . . 122

Chapter 5 Conclusion and Future Work 123
5.1 Summary . . . 123
5.2 Future Work . . . 125

Appendix A Benchmark Instances for TSP 127

Appendix B Benchmark Instances for CTSPP **129**

Appendix C 'Difficult' Cases on Two Rays for CTSPP **130**

Bibliography **133**

List of Tables

2.1 Comparison of average gap $OPT\%$ between two pairs from the Second Procedure using K-DENN 32
2.2 Comparison of results of average excess over optimal solutions: $OPT\%$ using Kalmanson heuristics (with best α) and classical versions 36
2.3 Comparison of results of average excess over HK lower bound: $HK\%$ using Kalmanson heuristics (with best α) and classical versions . . . 37
2.4 Number of instances improved by Kalmanson heuristics (with best α) compared with classical counterparts 37
2.5 Comparison of average number of crossings on benchmark instances using Kalmanson heuristics and classical counterparts 38

3.1 Comparison of tour construction for GRASP ($\alpha = 2$) and IGRASP ($\beta = 2$) 48
3.2 Experimental results of seven classical simple heuristics combined with three tour construction heuristics on self-generated random uniform instances: average solution quality and running time for each instance set 64
3.3 Experimental results of seven classical simple heuristics combined with three tour construction heuristics on self-generated random clustered instances: average solution quality and running time for each instance set 65
3.4 Experimental results of proposed algorithms and GRASP-2-opt with single-start on benchmark instances: average solution quality and running time for each instance set 67
3.5 Comparison of results of proposed algorithms and GRASP-2-opt on benchmark instances with controlled running time 69

3.6 Experimental results of the proposed algorithms and GRASP-2-opt with single-start on unweighted-tree instances: average solution quality and running time for each instance set 71
3.7 Comparison of results of proposed algorithms and GRASP-2-opt on unweighted-tree instances with controlled running time 72
3.8 Experimental results of the proposed algorithms and GRASP-2-opt with single-start on weighted-tree instances: average solution quality and running time for each instance set 72
3.9 Comparison of results of proposed algorithms and GRASP-2-opt on weighted-tree instances with controlled running time 73

4.1 Simple example of the Path TSP on the convex hull: coordinates of nodes 100
4.2 Simple example of the CTSPP on a two-ray case: coordinates of nodes 109
4.3 Comparison of results of Line heuristic and GRASP-2-opt with single-start on two-line instances 116
4.4 Comparison of results of Up-Down heuristic and GRASP-2-opt (with/without controlling running time) on convex-hull instances . . 117
4.5 Comparison of results of Up-Down heuristic and GRASP-2-opt (with/without controlling running time) on close-to-convex-hull instances 117
4.6 Experimental results of Two-Ray heuristic on size-24 two-ray instances: number of instances solved to optimality 119
4.7 Experimental results of Two-Ray heuristic on size-30 and -50 two-ray instances: number of instances improved by Two-Ray heuristic compared with RIH-$S(3,1)$ 120

List of Figures

2.1 Diamond instance 14
2.2 Comparison of two procedures on K-DENN: average gap $OPT\%$ with changing P for C1K 30
2.3 Comparison of two procedures on K-DENN: average gap $OPT\%$ with changing P for E1K 31
2.4 Effects of α on average gap for K-DENN on self-generated instances 33
2.5 Effects of α on average gap for K-NN on self-generated instances . . 33
2.6 Effects of α on average gap for K-GREEDY on self-generated instances 33
2.7 Comparison of results of optimal insertion of $first$ and placing $first$ in front for K-NN with $\alpha = 0.5$ 34
2.8 Comparison of DENN and K-DENN($\alpha = 0.6$) on benchmark instances 35
2.9 Comparison of NN and K-NN($\alpha = 0.5$) on benchmark instances . . . 35
2.10 Comparison of GREEDY and K-GREEDY($\alpha = 0.7$) on benchmark instances 36
2.11 Demonstration of crossing feature: the tour of instance kroB100 using DENN 38
2.12 Demonstration of crossing feature: the tour of instance kroB100 using K-DENN 38

3.1 Demonstration of 2-opt 49
3.2 Demonstration of 3-opt 50
3.3 Demonstration of chains neighbourhood 56
3.4 Demonstration of the first phase of the sliding window aggregation with $S_1 = \{x_2, x_3, x_4\}$, and $S_2 = \{x_6, x_7, x_8\}$ 58
3.5 Outcome of redefining S_2 with $m = 1$ in the second phase 58
3.6 Experimental results on 50-size self-generated uniform instances: trade-off between solution quality and running time for 21 algorithms 66

3.7 Comparison of results of proposed algorithms and GRASP-2-opt on benchmark instances with single-start: (a) 50-size instances; (b) 100-size instances; (c) 200-size instances; (d) 500-size instances 68
3.8 Performance of RIH-$S(2,1)$ with increasing number of starts on 50-size benchmark instances 70
3.9 Comparison of performance of RIH-$S(2,1)$ and RIH-$S(3,1)$ with increasing number of starts on 50-size benchmark instances 70
3.10 Comparison of solution gap plotted against running time for RIH-$S(2,1)$ and RIH-$S(3,1)$ on 50-size benchmark instances 71

4.1 Illustration of the classical Line-CTSPP: node distribution and path structure 78
4.2 Illustration of positions of the first violated pair (s_j, s_i) in τ^t, which is used in the TI-technique on Line-CTSPP 80
4.3 Illustration of Line-CTSPP with a fixed exit: distribution of nodes along a line 82
4.4 Illustration of four possible relative positions of the first violated pair (s_j, s_i) and exit point e in τ^t in the TI-technique on Line-CTSPP with a fixed exit 83
4.5 Illustration of the Part CTSPP Line with a fixed exit: distribution of nodes along a line 85
4.6 Illustration of the node distribution along a convex hull 95
4.7 Demonstration of V_i^1, where V_n^1 is the length of the clockwise-numbered path 96
4.8 Demonstration of V_2^j, where V_2^3 is the length of the anticlockwise-numbered path 97
4.9 Demonstration of the first case of V_i^j: Visit i directly from L^i 98
4.10 Demonstration of the second case of V_i^j: Visit i directly from U^i . . 98
4.11 Numerical example of Path TSP on a convex hull: (a) optimal path 1; (b) optimal path 2 102
4.12 Illustration of the node distribution along two rays 104
4.13 Illustration of $U(i,j,e)$, the cost of the lowest-cost sub-path through $1,2,\ldots,i-1,i,j,j+1,\ldots,n$, ending at e in an upper interval . . . 106
4.14 Illustration of $L(i,j,e')$, the cost of the lowest-cost sub-path through $1,2,\ldots,i-1,i,j,j+1,\ldots,n$, ending at e' in a lower interval 107
4.15 Illustration of node distribution of the upper interval $UI(a,b,d,e,p)$ 111

4.16 Comparison of one convex-hull instance and the corresponding close-to-convex-hull instance with n=20: (a) convex-hull instance; (b) close-to-convex-hull instance 113
4.17 Comparison of exact optimal route with Two-Ray heuristic route on 'Difficult-1' case that can be solved by a more sophisticated heuristic: (a) exact optimal route; (b) Two-Ray heuristic route. 119
4.18 Comparison of exact optimal route with Two-Ray heuristic route on 'Difficult-2' case that can be solved by a more sophisticated heuristic: (a) exact optimal route; (b) Two-Ray heuristic route 120
4.19 Comparison of exact optimal route with Two-Ray heuristic route on a 'More difficult' case that cannot be solved by a more sophisticated heuristic: (a) exact optimal route; (b) Two-Ray heuristic route . . . 121

Acknowledgments

I would like to express my utmost sincere gratitude to my supervisors Dr Vladimir Deineko and Dr Xuan Vinh Doan, who introduced me to the world of academic research on operational research. Their insightful ideas and guidance helped point my research into the right direction. It has been the greatest of pleasures to work with them, and without their unfailing support this thesis would probably not have been possible. I have been very fortunate to have such nice supervisors who have consistently guided and supported me in my academic research and daily life.

I would also like to express my sincere appreciation to Dr Bo Chen, Dr Juergen Branke, and Dr Xin Fei for their invaluable perspectives and productive discussions during my research. Special gratitude goes to my friend Mr Lixun Song for his support in programming.

I am particularly grateful to my parents Mr Shunkui Wang and Mrs Yanqin Sun, who supported me throughout my PhD study both financially and mentally. It is their encouragement and trust that have allowed me to resolutely follow my dreams. I would also like to express special thanks to my friends Dr Shiyuan Liu and Mrs Jiabei Zhang. I would not be where I am now without their encouragement. Finally, I am grateful to my fiance Dr Xing Wei for his exceptional support and love.

Declarations

This thesis is submitted to the University of Warwick in support of my application for the degree of Doctor of Philosophy. This thesis is my original work and has not been submitted, in whole or in part, for a degree at this or any other university.

Abstract

This thesis proposes heuristics motivated by solvable cases for the travelling salesman problem (TSP) and the cumulative travelling salesman path problem (CTSPP). The solvable cases are investigated in three aspects: specially structured matrices, special neighbourhoods and small-size problems. This thesis demonstrates how to use solvable cases in heuristics for the TSP and the CTSPP and presents their promising performance in theoretical research and empirical research.

Firstly, we prove that the three classical heuristics, nearest neighbour, double-ended nearest neighbour and GREEDY, have the theoretical property of obtaining the permutation for permuted strong anti-Robinson matrices for the TSP such that the renumbered matrices satisfy the anti-Robinson conditions. Inspired by specially structured matrices, we propose Kalmanson heuristics, which not only have the theoretical property of solving permuted strong Kalmanson matrices to optimality for the TSP, but also outperform their classical counterparts for general cases.

Secondly, we propose three heuristics for the CTSPP. The pyramidal heuristic is motivated by the special pyramidal neighbourhood. The chains heuristic and the sliding window heuristic are motivated by solvable small-size problems. The experiments suggest the proposed heuristics outperform the classical GRASP-2-opt on general cases for the CTSPP.

Thirdly, we conduct both theoretical and empirical research on specially structured cases for the CTSPP. Theoretically, we prove the solvability of Line-CTSPP on more general cases and the time complexity of the CTSPP on SUM matrices. We also conjecture that the CTSPP on two rays is NP-hard. Empirically, we propose three heuristics, which perform well on specially structured cases. The Line heuristic, based on Line-CTSPP, performs better than GRASP-2-opt when nodes are distributed on two close parallel lines. The Up-Down heuristic is inspired by the Up-Down structure in solvable Path TSP and outperforms GRASP-2-opt in convex-hull cases and close-to-convex-hull cases. The Two-Ray heuristic combines the path structures in the first two heuristics and obtains high-quality solutions when nodes are along two rays.

Chapter 1

Introduction and Related Works

1.1 Review of Problems and Background

Transportation plays a central role in modern society. Economically speaking, transportation accounts for between 6% and 12% of the GDP in many developed economies [132]. Transportation is also responsible for a large proportion of greenhouse gas emissions. Rodrigue [132] reported that transportation accounts for between 20% and 25% of energy consumption among developed economies, approximately 29% of world energy demand and about 61.5% of global oil consumption each year. The direct costs of transportation have increased dramatically since 2000 because of the significant increases in oil prices and the number of vehicles. This industry also imposes indirect costs on society including traffic congestion, death and injury, pollution and delay time costs [28].

Transportation is an important component of scientific research in the area of logistics. The Council of Supply Chain Management Professionals defines logistics as the process of planning, implementing and controlling the transportation and storage of goods and services from the depot to the point of consumption [146]. Routing problems in logistics define a series of combinatorial optimisation problems that optimise routes for a fleet of vehicles. Coelho et al. [42] conducted a survey on the applications of routing problems in logistics including oil transportation, retail, waste collection management, mail and package delivery and food distribution, and they demonstrated that optimisation techniques can result in significant economic savings of more than 10%. Research by Hasle and Kloster [84] also suggests that

the optimisation of routing can bring savings of 5–30%. There are two classical routing problems in logistics optimisation: the vehicle routing problem (VRP) and the travelling salesman problem (TSP), which are also central to combinatorial optimisation. The application of routing problems can help logistics provide better services while reducing costs and energy consumption.

The VRP is to optimise the routing design for a fleet of vehicles to serve a set of customers. Dantzig and Ramser [46] formulated a mathematical model for the VRP. The VRP is a generalisation of the multiple TSP. Hundreds of studies have analysed the VRP over the past 50 years. Exact methods aim to obtain optimal solutions. The classical exact algorithms include branch and X (X =cut, bound, price, etc.), which solve the VRP via integer linear programming (ILP) or mixed ILP (MILP) [13], dynamic programming, which recursively divides the original problem into smaller sub-problems [39], constraint programming, which interrelates different variables using constraints [145], and the column generation method, which breaks the problem into the master problem and the sub-problem [57].

The VRP is an NP-hard problem. This implies that the performance of exact algorithms on larger-size problems can be non-ideal. Therefore, a variety of heuristics have been developed. The well-known heuristics include the saving algorithm proposed by Clarke and Wright [41], the sweep algorithm developed by Gillett and Miller [71], the route-first-cluster-second algorithm [115] and the cluster-first-route-second algorithm [64]. In recent years, metaheuristics for the VRP have been investigated. Glover [74] coined the term 'metaheuristic', which refers to as 'heuristic guiding other heuristics'. Metaheuristics can also be explained as a strategy that combines subordinate heuristics to systematically explore different neighbourhood structures. Classical metaheuristics include nature-inspired metaheuristics such as the genetic algorithms (GA), ant colony system (ACS) and particle swarm optimisation (PSO), as well as memory-based metaheuristics such as the tabu search (TS). More heuristics and metaheuristics can be seen in the survey by Laporte et al. [100].

In addition to the classical VRP, some variants have been studied that incorporate more parameters and constraints such as a capacity constraint, a length limit of the route, an arrival/departure time limit, the collection or delivery of goods and service time requirements. The well-known variants include capacitated VRP (CVRP, [41]), VRP with split delivery (VRPSD, [56]), VRP with back-haul (VRPB, [76]), multi-depot VRP (MDVRP, [101]), dynamic VRP [70] and stochastic VRP [69]. The latest advances and new challenges in the VRP have been

extensively discussed by Toth and Vigo [141] and Golden et al. [78].

The TSP is another intensively studied problem in logistic optimisation. The TSP is to find the optimal tour to minimise the total travelling distance for one vehicle that visits given customers exactly once and then returns to the starting point. The TSP was first put forward by a travelling salesman in a German book in 1832, and a mathematical model was first formulated by Flood (1956) [65]. Since then, the problem has aroused great interest in academia because it is a theoretically challenging problem with important practical applications in many areas such as transportation, telecommunications, logistics, manufacturing and neuroscience. Details of applications of the TSP have been given by Davendra [47].

The TSP is also an NP-hard problem. During the past few decades, numerous exact algorithms, approximation algorithms, heuristics and metaheuristics have been explored for the TSP. Lawler [104] provided an insightful and comprehensive survey of algorithms for this problem. We summarise the major exact algorithms in this thesis. The well-known exact algorithms include ILP [45, 109, 99], the branch and bound algorithm [31, 11, 110] and the minimum spanning tree bound algorithm [38, 85].

A large-size TSP is computationally intractable. Therefore, a more practical approach is to develop heuristics to find nearly optimal solutions within a reasonable computation time. Heuristics include those with guaranteed worst-case performance and those with good empirical performance. Examples of the former heuristics can be seen in [4, 121], but the majority of researchers have focused on the use of the latter heuristics to improve the performance. TSP heuristics can also be grouped into tour construction heuristics, tour improvement heuristics and composite heuristics. Tour construction heuristics construct the tour incrementally until a valid tour is completed. The well-known tour construction heuristics include the nearest neighbour algorithm [133], double-ended nearest neighbour algorithm [17], insertion algorithm [133, 139, 116] and patching algorithm [93]. Tour improvement heuristics improve the current tour repeatedly by performing various operations. The r-opt algorithm (Lin-Kernighan heuristic [105, 106]), simulated annealing [94] and the TS [73] are well-researched tour improvement heuristics. There are also two effective composite algorithms: the CCAO heuristic [77] and GENIUS algorithm [68]. Heuristics have been compared by Bentley [17], Reinelt [128, 129] and Johnson and McGeoch [89]. We refer readers to Applegate et al. [6] and Gutin and Punnen [81] for more information on the TSP.

In the literature, different variants of the TSP have been investigated: profit-

based TSP [61], time-window-based TSP [136], maximum TSP [15] and kinetic-based TSP [36].

There is a special version of the TSP, the Path TSP, the objective function of which is the same as the TSP, but the vehicle does not return to the start point, which means that the tour is a path instead of a cycle as in the classical TSP. The field of research into the Path TSP is less rich than that for the classical TSP. The Path TSP is also an NP-hard problem. Recently, Zenklusen [149] explored a 1.5-approximation algorithm for the Metric Path TSP.

The cumulative travelling salesman path problem (CTSPP [18]) is another variant of the classical TSP. In the literature, the CTSPP is also known as the minimum latency problem (MLP [20]), the travelling repairman problem (TRP [142]), the delivery man problem [63] and the school bus driver problem [37].

The CTSPP is similar to the TSP in that it involves visiting each customer exactly once, but the objective is to minimise the sum of arrival times at all customers instead of the total travelling time in the classical TSP. Note that for the CTSPP, the tour is a path as in the Path TSP. The cumulative objective function has arisen from the increasing incidence of natural disasters worldwide, in which the transportation cost can be negligible compared with the value of people's lives. After a disaster, emergency relief should be delivered to each affected area as soon as possible; thus, the objective is to minimise the sum of waiting times for all areas [16]. Considering the current COVID-19 pandemic, the application of the CTSPP to the delivery of medical supplies has raised concerns. Given that medical supplies are delivered to cities with different numbers of people, we assume that there is a weight associated with each city. In this case, the objective is changed to minimising the sum of arrival times at all cities with weights. We call the new problem the CTSPP with weights (CTSPPW). The CTSPP is a subset of the CTSPPW in which the weights for all cities are 1.

The CTSPP considers the satisfaction of each customer, and thus has many important practical applications in customer-centric service systems such as in logistics for humanitarian relief supply [29], school bus routing [24] and home delivery services [108]. In contrast, in the classical TSP, minimising the total travelling time may result in some customers experiencing a severe delay, leading to very long total waiting times. On account of this, Archer and Williamson [8] stated that the objective of the CTSPP is customer oriented instead of server oriented, as in the TSP. Wu et al. [148] stated that the CTSPP considers the service quality, while the TSP considers the service cost. The CTSPP can also be applied to other practical

models such as disk-head scheduling [20], data retrieval [59], job scheduling and different-location machine repair [114].

The CTSPP is an NP-hard problem similarly to the TSP but, surprisingly, the former is a more difficult problem than the latter [20, 108]. The CTSPP has special properties that do not exist in the TSP. One property is that very small local changes can produce highly non-local changes in the optimal tour [20]. Another property is that an additional edge inserted at the start of the route will have a huge effect on the waiting time of all remaining customers [10]. The literature suggests that the CTSPP is computationally much harder to solve to optimality or even approximately than the TSP [108].

Although the TSP has been intensively studied in the area of combinational problems, the CTSPP has received insufficient attention. Some exact algorithms have been proposed for the CTSPP such as enumerative algorithms based on non-linear integer programming [18, 107], enumerative algorithms that incorporate lower bounds from linear integer programming [63], dynamic programming algorithms [147], branch and bound algorithms [148, 19] and branch-cut-and-price formulations [2]. However, these exact algorithms can only be applied to small-size problems. For example, Wu et al. [148] solved a small-size problem of 25 within 100 seconds despite using a more efficient algorithm than the standard formulation of CPLEX. Approximation algorithms for the CTSPP are given in [20, 37, 10, 60, 7]. The best-known approximation factors are 3.59 for general cases [37] and 3.03 for the edge-weighted trees [7], which are greater than the current best approximation factor of 1.5 for the TSP. Metaheuristics that adopt the greedy randomised adaptive search procedure (GRASP [62]) for tour construction and adopt the variable neighbourhood descent (VND) or variable neighbourhood search (VNS) for tour improvement have also been explored [134, 137].

1.2 Solvability of TSP-Related Problems

Although the TSP and the CTSPP are NP-hard problems, special solvable cases exist. In the literature, the solvability of the TSP and the CTSPP has been studied in three aspects: specially structured matrices that can be solved in polynomial time, exponential neighbourhoods that can be searched in polynomial time and mathematical programming models for small-size problems. Solvable cases are not only useful in their own right, but also useful in heuristics for more general cases [40]. This motivates our research on the heuristics inspired by solvability in

this thesis.

Our emphasis will be on how solvability can be applied to heuristics for the TSP and the CTSPP. We propose heuristics inspired by the solvable cases and demonstrate their promising performance. We outline each type of solvability as follows.

Solvable Specially Structured Matrices

This type of solvable special case has underlying distance matrices with special combinatorial properties that are formulated in terms of specific inequalities. Given such a specifically structured matrix, the basic idea is to first investigate the combinatorial structure of an optimal solution and show that there is always an optimal solution having certain properties or a special structure. Thus, the original problem can be simplified to an easier problem of finding the optimal tour among those tours with certain properties or special structures. Next, we show that we can solve to optimality over the tours with the specific structure in polynomial time, and we can say that the problem on such a special matrix is polynomially solvable.

Typically, TSP problems with distance matrices such as the Supnick matrix [140], Kalmanson matrix [92], Monge matrix [48], Demidenko matrix [54] and Van der Veen matrix [144] can be solved using polynomial algorithms. Deineko et al. (2014) classified the specially structured cases for the TSP under four-point conditions. Çela et al. [33] explored the x-and-y-axes TSP where all cities are located on the x-axis and the y-axis of an orthogonal coordinate system of the Euclidean plane, and solved the problem in quadratic time. Burkard et al. [27] also described several specially solvable classes of Euclidean instances of the TSP with special geometric properties such as the TSP on convex sets, the TSP on line segments and the necklace TSP. We refer readers to the well-known surveys by Lawler [104] and Burkard et al. [27] for solvable cases with specially structured matrices for the TSP.

The specially structured matrices mentioned above depend on the numbering of cities, and the specially structured conditions can be checked in polynomial time. The problem becomes non-trivial if a given distance matrix does not satisfy the specially structured conditions but can be permuted into a matrix that satisfies certain conditions by renumbering rows and columns in the initial matrix. In this case, the initial matrix is called a permuted matrix. A recognition algorithm is to determine whether there exists a renumbering, i.e., a permutation of the rows and columns of the original matrix such that the renumbered matrix satisfies the spe-

cific conditions, and to obtain the permutation in the case that it exists. Permuted Monge matrices, permuted Supnick matrices and permuted Kalmanson matrices can be recognised in polynomial time [48, 50, 55]. However, the recognition algorithms for permuted Demidenko matrices and permuted Van der Veen matrices have not been resolved.

For the Path TSP, Garcia and Tejel [67] proposed a polynomial-time algorithm for a specially structured case where all nodes on the Euclidean plane are along the border of their convex hull. Çela et al. [35] solved the Path TSP with a Demidenko matrix in polynomial time. For the CTSPP, algorithms also exist for specially structured cases such as paths where nodes are on a straight line [3], edge-unweighted trees [111], three-diameter trees [20] and trees with a constant number of leaves [98], which can all be solved in polynomial time.

Although some of the solvable cases mentioned above do not have a specific matrix name, they are a subclass of some specially structured matrices. When nodes are on a straight line, the underlying matrix is an anti-Robinson matrix [131]. Çela et al. [35] suggested that given an edge-weighted tree, if the nodes are numbered with the depth-first route, then the shortest path distances determine a Kalmanson matrix. In addition, any convex-hull case satisfies Kalmanson conditions.

We have found that in current research on specially structured cases for both the TSP and the CTSPP, most of the effort has been devoted to theoretical studies. Relevant heuristics that utilise this type of solvable cases are scarce in the literature. In this thesis, we will propose heuristics motivated by the specially structured matrices for both the TSP and the CTSPP. We will also investigate new specially structured cases to contribute to the theory and explore more heuristics for special cases and more general cases.

Solvable Neighbourhoods

Neighbourhood search algorithms are a broad class of improvement heuristics where an improved solution is found at each iteration by searching the neighbourhood of the current solution. To be more specific, a neighbourhood search algorithm is an iterative procedure that starts with an initial solution and searches its neighbourhood to find the best neighbour, i.e., the one with the best objective value. If the best neighbour improves the solution, then the current solution is replaced by the best neighbour, and the neighbourhood search is performed again. The process is repeated until there is no further improvement [51]. The returned solution is a locally optimal solution. The neighbourhood search is widely used for

the TSP and other combinatorial optimisation problems [1].

If the size of the neighbourhood is polynomial in the input size, we can search the whole neighbourhood by enumerating all neighbours in polynomial time. However, when the neighbourhood size is small, only a small number of candidates can be searched for each iteration, which may lead to a very large number of iterations. Therefore, it is necessary to explore exponential neighbourhoods that can be searched in polynomial time [51].

Exponential neighbourhoods have been extensively studied [12, 82, 120]. We summarise the well-known neighbourhoods here. The ASSIGN neighbourhood [135] contains $(n/2)!$ neighbours and can be searched in $O(n^3)$ time. Based on assignments, Gutin [80] generalised this neighbourhood, which can also be searched in $O(n^3)$ time. Punnen [127] extended the neighbourhood further by removing and reinserting paths instead of nodes. Deineko and Woeginger [51] also explored neighbourhoods based on matches in bipartite graphs and neighbourhoods based on partial orders.

Permutation trees are another group of exponential neighbourhoods that can be searched in polynomial time [51]. A permutation tree is a rooted ordered tree with pairwise distinct leaves. The PQ-tree is a classical permutation tree [22]. In addition, the well-known pyramidal neighbourhood [54] is also represented by a permutation tree. A permutation is called a pyramidal permutation when some of the nodes are placed in increasing order to n and the remaining nodes are placed in decreasing order. Although the pyramidal neighbourhood contains an exponential number (2^{n-1}) of pyramidal permutations, an optimal pyramidal permutation can be obtained within $O(n^2)$ time using dynamic programming [54] for the TSP. The pyramidal neighbourhood is also related to some well-known specially structured distance matrices for the TSP. Demidenko [54] proved that the TSP with a Demidenko matrix is pyramidally solvable, which means that an optimal tour for the TSP can be found among tours with pyramidal structures. In the survey [27], Burkard et al. summarised the pyramidally solvable TSPs. A moderate number of matrices such as Supnick matrices, Kalmanson matrices, Demidenko matrices and Van der Veen matrices have a combinatorial structure that guarantees that the optimal solution can be obtained in the pyramidal neighbourhood. The solvability can be proved by a tour improvement technique [27].

Sarvanov and Doroshko [135] first applied the pyramidal neighbourhood to a local search algorithm for the TSP. Carlier and Villon [30] combined the pyramidal neighbourhood with a cyclic shift, and the resulting local search heuristic had much

better performance than 2-opt and was even competitive with k-opt [30]. We also refer readers to surveys by Deineko and Woeginger [51] and Ahuja et al. [5] for a wealth of information on exponential neighbourhoods that can be searched in polynomial time.

We have found that although the TSP on solvable neighbourhoods has been intensively studied, the CTSPP has received insufficient attention. In our thesis, we will explore heuristics based on solvable neighbourhoods for the CTSPP.

Solvable Size

Due to the NP-hardness, the exact algorithms can only be applied to small-size problems. However, there is also a decomposition approach [14], which can divide a large problem into a sequence of sub-problems such that each sub-problem can be solved to optimality by mathematical programming. This type of heuristic is a class of matheuristics [23], which combine heuristics with mathematical programming models to obtain high-quality solutions for large-size problems. Matheuristics can be applied to different routing problems and are classified into three types: decomposition approaches, improvement heuristics using mathematical programming and branch-and-price (or column-generation-based) approaches. More information concerning decomposition heuristics for the TSP can be found in publications by Ball [14] and Archetti and Speranza [9]. However, in the literature, related research on the CTSPP is scarce. In this thesis, we will explore efficient heuristics that use strategies to reduce the problem size and apply dynamic programming algorithms to the solvable-size problems for the CTSPP.

1.3 Structure of Thesis and Contributions

The next chapters of this thesis propose heuristics motivated by solvable cases for the TSP and the CTSPP. The brief summary of each chapter and its contributions are given below.

Chapter 2: Kalmanson Heuristics for the TSP. In this chapter, we amend three classical heuristics motivated by specially structured matrices for the TSP. We consider three classical heuristics, nearest neighbour (NN), double-ended nearest neighbour (DENN) and GREEDY, for which we prove their theoretical property of obtaining the permutation for a permuted strong anti-Robinson matrix such that the renumbered matrix satisfies the anti-Robinson conditions. The anti-Robinson matrix is closely related to and can be obtained after a simple transformation of the Kalmanson matrix. Inspired by the knowledge of the Kalmanson matrix and

the anti-Robinson matrix, we suggest minor amendments to the three classical heuristics. The amended versions, called Kalmanson heuristics, exhibit surprising results, including not only the additional theoretical property of obtaining the permutation for a permuted strong Kalmanson matrix, and thus finding an optimal solution for the TSP with a permuted strong Kalmanson matrix, but also superior empirical performance to their classical counterparts for general cases based on extensive computational experiments. The incorporation of additional features from solvable cases into the classical heuristics enriches the nature of the heuristics.

Chapter 3: Simple Heuristics for the CTSPP. In this chapter, we propose three heuristics based on dynamic programming techniques for tour improvement. The first heuristic, the pyramidal heuristic (PH), is motivated by the well-known solvable pyramidal neighbourhood. We develop dynamic programming recursions to find the best permutation in the pyramidal neighbourhood for the CTSPP. The second heuristic, chains heuristic (CH), and the third heuristic, the sliding window heuristic (SWH), are motivated by solvable small-size problems. We use different strategies to reduce the problem size in the CH and SWH. The CH partitions the tour into a set of small-size chains and organises a chains neighbourhood, while the SWH adopts a sliding window aggregation strategy with intervals to represent a new customer. We also propose two tour construction heuristics: IGRASP and RIH. We provide detailed analyses and compare the effectiveness of various simple heuristics by conducting extensive computational experiments. The proposed algorithms outperform the classical GRASP-2-opt, especially when they are applied to specially structured case, weighted trees. The experiments suggest that the dynamic-programming-based heuristics motivated by solvable cases can obtain promising results for the CTSPP. In addition, the experiments can give general insights into the better selection and combination of simple heuristics in the future study of metaheuristics.

Chapter 4: Heuristics for the Special CTSPP. This chapter describes both theoretical and empirical research on specially structured cases for the CTSPP. Theoretically, we extend the solvability of Line-CTSPP by fixing the exit node and considering the straight line as part of the total path. We show that the extended cases can be solved with a dynamic programming algorithm in polynomial time. Also, we prove the solvability of the CTSPPW with all nodes on a straight line. In addition, we prove that the time complexity of the CTSPP on the SUM matrix is $O(nlogn)$, and we conjecture that the CTSPP on two rays (a special subclass of the convex hull) is NP-hard. Empirically, inspired by the solvable cases,

we propose dynamic-programming-based heuristics for the CTSPP. We conduct computational experiments to show that the proposed heuristics perform well on specially distributed cases. The Line heuristic is based on the path structure in the solvable Line-CTSPP and displays superior performance to the classical GRASP-2-opt in both running time and solution quality when the nodes are distributed on two close parallel lines. The Up-Down heuristic is inspired by the Up-Down structure in the solvable Path TSP. This heuristic outperforms the classical GRASP-2-opt in convex-hull cases and close-to-convex-hull cases. The Two-Ray heuristic combines the path structures in the Line heuristic and the Up-Down heuristic and obtains high-quality solutions when nodes are specially distributed along two rays.

Chapter 5: Conclusion and Future Work. This chapter provides concluding remarks and discusses potential future research areas.

1.4 Notation

This section gives the notation used in this thesis.

Given a weighted graph $G = (V, E, C)$, $V = \{v_1, \ldots, v_n\}$ is a vertex set, in which each vertex represents the location of one customer, and v_1 represents the depot. We define an $n \times n$ distance matrix $C = (c_{ij})$, where $c_{v_i v_j}$ denotes the travelling cost between v_i and v_j for each edge $(v_i, v_j) \in E$. We also define a tour $\tau = \langle \tau_1, \tau_2, \ldots, \tau_n, \tau_{n+1} = \tau_1 \rangle$ for the TSP and a tour $\tau = \langle \tau_1, \tau_2, \ldots, \tau_n \rangle$ for the Path TSP, CTSPP and CTSPPW, where τ_i is the i^{th} vertex in the tour.

In a neighbourhood search, a permutation is denoted as $\pi = \langle \pi_1, \ldots, \pi_i, \ldots, \pi_n \rangle$, where π_i is the i^{th} index in this permutation. Here, for a tour τ, $\tau(\pi)$ is its neighbour and $N(\tau) = \{\tau(\pi) : \pi \in N\}$ is a specific neighbourhood of τ, assuming that N is the set of all permutations that satisfy a specific condition.

Chapter 2

Kalmanson Heuristics for the TSP

2.1 Introduction and Related Works

The TSP can be defined mathematically as follows. Given an $n \times n$ distance matrix $C = (c_{ij})$, where c_{ij} denotes the travelling distance between customer i and customer j, the objective is to find a tour $\tau = \langle \tau_1, \tau_2, \ldots, \tau_n, \tau_{n+1} = \tau_1 \rangle$ with the minimum travelling distance, where the set $\{\tau_1, \tau_2, \ldots, \tau_n\}$ is the set of locations $\{1, 2, \ldots, n\}$. We assume that all distance matrices considered in this chapter are symmetric. Since the tour is uniquely defined by a permutation of nodes, in what follows, the terms *permutation* and *tour* will both be used interchangeably if the meaning is clear from the context. The length of the tour τ is denoted as $c(\tau)$, which can be calculated using the following function.

$$c(\tau) = \sum_{i=1}^{i=n} c_{\tau_i \tau_{i+1}} \tag{2.1}$$

In this chapter, three well-known classical construction heuristics for the TSP are considered: Nearest Neighbour (NN), Double-ended Nearest Neighbour (DENN) and GREEDY. They are all pure augmentation heuristics which are used to construct tours by adding one edge at a time based on the length of the edges. We refer readers to Section 3.2 of Johnson and McGeoch [90] for a detailed description of the above algorithms and the results of an empirical investigation. These heuristics are also analysed in [133, 66, 118, 88, 87].

Our objective is to combine the simplicity of the NN, DENN and GREEDY heuristics with the knowledge of specially structured matrices to enrich the nature of heuristics so that they can not only solve to optimality the TSP in some special cases, but also improve the performance of the heuristics for general cases.

The proposed heuristics are inspired by the Kalmanson matrix [92]. A *symmetric* $n \times n$ matrix $C = (c_{ij})$ is called a *Kalmanson* matrix if it fulfils the Kalmanson *conditions*

$$c_{ij} + c_{lm} \leq c_{il} + c_{jm}, c_{im} + c_{jl} \leq c_{il} + c_{jm}, \text{ for all } 1 \leq i < j < l < m \leq n. \quad (2.2)$$

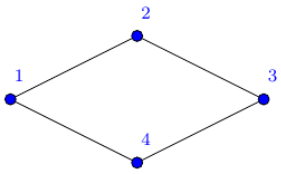
These conditions are called quadrangle inequalities [92]. These inequalities capture the property of convex quadrangles that the sum of the lengths of the two diagonals is always greater than or equal to the sum of the lengths of the two opposite sides. Kalmanson [92] proved that the TSP with a distance matrix satisfying the conditions 2.2 is solved to optimality by the tour τ defined by the identity permutation $\tau = \langle 1, 2, 3, \ldots, n-2, n-1, n, 1 \rangle$. He also suggested that if n points are located on the boundary of the convex hull and numbered along the convex hull, the corresponding matrix fulfils the Kalmanson conditions.

In combinatorial optimisation, a cyclic shift is an operation that rearranges the entries in a permutation by either moving the last element to the first position and moving the other elements to the following positions or by executing the inverse operation. Given a permutation containing n elements, the cyclic shift can be repeatedly executed to produce $n-1$ permutations. When the points are renumbered after a cyclic shift, the Kalmanson conditions still hold; thus, the tours obtained from a cyclic shift of the tour τ are also optimal solutions. Moreover, conditions 2.2 still hold when positive or negative constants are added to rows and columns in the distance matrix.

The Kalmanson matrix is closely related to the master tour problem [55]. For the TSP, the master tour has a special property that the sub-tour obtained from the master tour retains its optimality for the remaining nodes when any subset of nodes is removed from the master tour. The master tour problem is to determine whether there exists a master tour given a distance matrix. Deineko et al. [55] proved that a distance matrix has a master tour if and only if it is a Kalmanson matrix. The special structure of the Kalmanson matrix can also be applied to other combinatorial optimisation problems. The relevant research can be seen in [52, 95, 43, 34, 122].

Clearly, conditions 2.2 can be checked in $O(n^2)$ time. If the matrix is indeed a Kalmanson matrix, then the optimal TSP solution is already known. The problem becomes non-trivial if a given distance matrix does not satisfy conditions 2.2 but can be permuted into a Kalmanson matrix by renumbering rows and columns in the initial matrix. In this case, we need a *recognition* algorithm to find the correct permutation of rows (and columns). The recognition algorithm is to determine whether there exists a renumbering, i.e., a permutation π of the rows and columns of the original matrix $C = (c_{ij})$ such that the renumbered matrix $C^\pi = (c^\pi_{ij}) = (c_{\pi(i)\pi(j)})$ satisfies the specific conditions and, if it exists, to obtain the permutation π. Note that for the permuted Kalmanson matrix, π is also an optimal solution for the TSP.

The classical NN, DENN and GREEDY heuristics cannot solve the TSP with permuted Kalmanson matrices to optimality. It is easy to construct an instance where nodes are along the boundary of their convex hull, i.e., the underlying matrix is a permuted Kalmanson matrix, but none of the three classical heuristics can find an optimal solution. A diamond instance is used to illustrate this fact in Figure 2.1. If NN, DENN or GREEDY is applied to the instance, edge $(2, 4)$ will always be included in the TSP tour, which is not optimal. The objective of this chapter is to modify these classical heuristics, thus enabling them to solve the TSP to optimality in some special cases.



$$D = (d_{ij}) = \begin{pmatrix} 0 & 13 & 24 & 13 \\ 13 & 0 & 13 & 10 \\ 24 & 13 & 0 & 13 \\ 13 & 10 & 13 & 0 \end{pmatrix}$$

Figure 2.1: Diamond instance

A Kalmanson matrix can be transformed into an anti-Robinson matrix [131]. A *symmetric* $n \times n$ matrix $A = (a_{ij})$ is called an *anti-Robinson* matrix if it satisfies the anti-Robinson *conditions*

$$a_{ik} \geq \max\{a_{ij}, a_{jk}\}, \quad \text{for all} \quad 1 \leq i < j < k \leq n. \tag{2.3}$$

These conditions mean that the entries in each row and column of an anti-Robinson matrix are non-decreasing when moving away from the main diagonal in any direction. Clearly, the maximal item of the matrix is in row 1 and column n. If points are sorted in an order such that the underlying matrix satisfies

Conditions 2.3, then the order is called a compatible order [123]. The Robinson matrix [131] was first used as a tool to solve the problem of chronologically ordering archaeological deposits, where the chronological order is one of the compatible orders. Other applications of (anti-) Robinson matrices can be seen in [123].

The anti-Robinson matrix and Kalmanson matrix are both subclasses of the so-called Demidenko matrix [54]. A *symmetric* $n \times n$ matrix $C = (c_{ij})$ is called a *Demidenko* matrix if it fulfils the *conditions*

$$c_{ij} + c_{lm} \leq c_{il} + c_{jm}, \text{ for all } 1 \leq i < j < l < m \leq n. \tag{2.4}$$

We now show the relationship between Kalmanson matrices and anti-Robinson matrices.

Lemma 2.1.1 *If a Kalmanson matrix C is transformed into the matrix $C' = (c'_{ij})$ with $c'_{ij} := c_{ij} - c_{i1} - c_{1j}$, then the $(n-1) \times (n-1)$ matrix obtained by deleting the first row and first column in C' is an anti-Robinson matrix.*

Proof. Given a Kalmanson matrix $C = (c_{ij})$, the matrix C' obtained by transforming C into a matrix with zeros in the first row and the first column by using $c'_{ij} = c_{ij} - c_{i1} - c_{1j}$ is also a Kalmanson matrix satisfying

$$\begin{aligned}
&(c_{ij} - c_{i1} - c_{1j}) + (c_{lm} - c_{l1} - c_{1m}) \leq (c_{il} - c_{i1} - c_{1l}) + (c_{jm} - c_{j1} - c_{1m}),\\
&(c_{im} - c_{i1} - c_{1m}) + (c_{jl} - c_{j1} - c_{1l}) \leq (c_{il} - c_{i1} - c_{1l}) + (c_{jm} - c_{j1} - c_{1m}),\\
&\text{thus, } c'_{ij} + c'_{lm} \leq c'_{il} + c'_{jm},\ c'_{im} + c'_{jl} \leq c'_{il} + c'_{jm}, \quad \text{for all } 1 \leq i < j < l < m \leq n.
\end{aligned} \tag{2.5}$$

In the matrix C', $c'_{ik} = 0$ for all $i = 1$, $1 \leq k \leq n$; thus, $c'_{lm} \leq c'_{jm}$, $c'_{jl} \leq c'_{jm}$, for all $1 < j < l < m \leq n$. Then the first row and the first column of C' are deleted to generate an $(n-1) \times (n-1)$ matrix, which satisfies $c'_{lm} \leq c'_{jm}$, $c'_{jl} \leq c'_{jm}$ for all $1 \leq j < l < m \leq n$; thus, the $(n-1) \times (n-1)$ matrix is an anti-Robinson matrix. This completes the proof of the lemma. □

Assume we are given a permuted Kalmanson matrix C for which we aim to find a permutation such that the renumbered matrix satisfies the Kalmanson conditions, i.e., to find the optimal TSP solution. Since after a cyclic shift of rows and columns in a Kalmanson matrix, the resulting matrix is still a Kalmanson matrix, the cyclic shift of a permutation will not change its property. Therefore, any index can be chosen to be the index $first$ in the permutation. Given the fixed

index $first$, the matrix is first transformed using $c'_{ij} = c_{ij} - c_{i,first} - c_{first,j}$ and the row $first$ and column $first$ are deleted, then the obtained $(n-1) \times (n-1)$ matrix C' is a permuted anti-Robinson matrix. In this case, the problem of obtaining the permutation (i.e., the optimal TSP solution) for a permuted Kalmanson matrix is now simplified to obtaining the permutation for a permuted anti-Robinson matrix. This suggests that if we can first find a permutation τ' for the transformed matrix C', then the optimal TSP solution to the initial matrix C will be the tour τ which starts with $first$ followed by the permutation τ'.

An intuitive idea for amending heuristics in the hope that they can solve a special case to optimality is to incorporate a recognition algorithm for this special case into the heuristics. However, all the recognition algorithms for Kalmanson matrices or anti-Robinson matrices in the literature are elaborate and difficult to incorporate into the existing heuristics [40, 55, 123]. The main difficulties arise in cases when many inequalities in conditions 2.2 and 2.3 are not strict. To obtain simple algorithms, we consider special subclasses of the matrices. A matrix is called a *strong* Kalmanson (anti-Robinson) matrix if it satisfies all inequalities in conditions 2.2 (2.3) with strict inequalities. Two examples are given to readers to give them an intuitive understanding of the structure of such matrices: one example is that if n points in the Euclidean plane are the corner points of their convex hull, then the corresponding distance matrix is a permuted strong Kalmanson matrix; the other example is that if n points are on a straight line, then the matrix is a permuted strong anti-Robinson matrix. In particular, in a strong anti-Robinson matrix, none of the rows (columns) contains more than two items with the same value.

The remainder of this chapter is structured as follows.

In Section 2.2, we prove that the three classical construction heuristics, NN, DENN and GREEDY, can be used to obtain the permutation for permuted strong anti-Robinson matrices such that the renumbered matrices satisfy the anti-Robinson conditions, which is a new theoretical property. We then amend the three heuristics motivated by the relationship between the Kalmanson matrix and anti-Robinson matrix in Lemma 2.1.1. These amended versions have the additional theoretical property of obtaining the permutation for a permuted strong Kalmanson matrix, which means that when the amended heuristics are applied to an instance with a permuted strong Kalmanson matrix, an optimal solution for the TSP can be found.

In Section 2.3, empirical investigations are conducted and the performance

of the proposed heuristics is demonstrated. The results of computational experiments on the benchmark instances from the TSP DIMACS Challenge [90] clearly indicate the superiority of the amended heuristics compared with the original counterparts for general cases.

Section 2.4 concludes this chapter.

2.2 Heuristics Motivated by Kalmanson Matrix

We first prove the theoretical property of the three classical construction heuristics, NN, DENN and GREEDY, of obtaining the permutation for permuted strong anti-Robinson matrices such that the renumbered matrices satisfy the anti-Robinson conditions. Motivated by the Kalmanson matrix, we then propose minor amendments to the three heuristics to give them an additional theoretical property.

2.2.1 Classical Heuristics and Theoretical Properties for the TSP

In the NN heuristic, the nearest customer is repeatedly visited until all customers are visited. In NN, the tour starts at a *start* node τ_1. At each step, a new node, the node closest to the most recently added node, is added to the permutation. The algorithm ends when all nodes are included in the permutation. Mathematically, for the partial tour $\langle \tau_1, \tau_2, \ldots, \tau_i \rangle$, we choose the node that is nearest to τ_i as the next node τ_{i+1} from the nodes not yet chosen until all nodes are included. Given an $n \times n$ matrix C, the tour can be completed within $O(n^2)$ time.

The DENN heuristic also uses the nearest neighbour rule to construct the tour. The difference between NN and DENN is that for NN, the nodes can only be added on one side to complete the tour, while for DENN, the nodes can be added on either side (left or right) to update the two endpoints of the partial tour. In DENN, an initial tour $\langle \tau_1, \tau_2 \rangle$ is constructed as in the first step of NN. Node τ_1 is referred to as the *left* node and τ_2 is referred to as the *right* node. The tour is extended by adding the nearest node (which is nearest to either the left or right node). The reference to the *left*/*right* is updated, and the process continues until the complete tour is constructed. Recursively, given the partial tour $\langle \tau_i, \ldots, \tau_1, \tau_2, \ldots, \tau_j \rangle$, from the nodes not yet chosen, we assume that τ_{i+1} and τ_{j+1} are the nearest neighbours to τ_i and τ_j respectively. We then compare the two lengths $c_{\tau_i \tau_{i+1}}$ and $c_{\tau_j \tau_{j+1}}$ and add the edge with the shorter length on the corresponding side and update the corresponding endpoint. The complexity of this heuristic is also $O(n^2)$.

The GREEDY heuristic starts with sorting all edges from the shortest to the longest. The tour is then constructed by sequentially adding the first eligible edge from the sorted list. An edge is eligible if its inclusion into a partially constructed tour neither yields a cycle nor results in a node having a degree greater than 2. In the end, when only two nodes with a degree of 1 remain in the partial tour (which is a path at this stage), the two nodes are linked together to form a tour (cycle). The complete tour can be constructed within $O(n^2 logn)$ time.

Next, the theoretical properties of the three classical heuristics are provided, which are not covered in the literature. We begin with DENN and its relevance to obtaining the permutation for permuted strong anti-Robinson matrices.

Lemma 2.2.1 *Given a* permuted *strong anti-Robinson matrix $A = (a_{pq})$, permutation τ constructed in DENN permutes the matrix into the strong anti-Robinson matrix.*

Proof. In the proof we will show that DENN places indices in a tour at positions that correspond to the unique positions of the indices in a permutation. In accordance with the DENN rule, we place nodes on the basis of inequalities. Also, from the structure of anti-Robinson matrices, the relative positions of nodes also indicate inequalities. These inequalities should not be violated.

Considering the TSP with a permuted strong anti-Robinson matrix $A = (a_{pq})$, we here reformulate the first step in DENN: we start with a start node s, find the node j nearest to s and define the initial tour as $\tau = \langle \ldots, s, j, \ldots \rangle$. In row s, a_{sj} is the smallest item.

From the anti-Robinson conditions, we know that for an anti-Robinson matrix, the entries in each row and column are non-decreasing when moving away from the main diagonal in any direction. Because a_{sj} is the smallest item in row s, if we want to permute rows and columns in the distance matrix to ensure that the items are non-decreasing away from the diagonal, then item j must be placed next to item s in the permutation, similar to the construction in DENN. If s must be the first item or the last item in the permutation, then index j is unique. Otherwise, two candidates, j_1 and j_2, may be placed next to s. According to the structure in the strong anti-Robinson matrices, possible orderings are $\langle \ldots, j_1, s, j_2, \ldots \rangle$ and $\langle \ldots, j_2, s, j_1, \ldots \rangle$.

The conditions of anti-Robinson matrices indicate that if A is an $n \times n$ strong anti-Robinson matrix, then $A^\sigma = (a_{\sigma(i)\sigma(j)})$ is also a strong anti-Robinson matrix with the permutation $\sigma =< n, n-1, n-2, \ldots, 2, 1 >$. Therefore, if the permutation

that corresponds to tour τ permutes a matrix into an anti-Robinson matrix, then its inverse, i.e., the permutation that corresponds to the inverse tour τ^{-}, can also be used as a permutation. As discussed above, item j must be placed next to item s in the permutation; therefore w.l.o.g. we can start with a permutation where s precedes j: $\tau = \langle \ldots, s, j, \ldots \rangle$.

In DENN, node s is set as the current *left* node, $L = s$, and node j is set as the current *right* node, $R = j$. In the next steps DENN finds index k: $a_{kL} = min_{q \neq L,R} a_{qL}$ and index $m : a_{Rm} = min_{q \neq L,R} a_{Rq}$. The node corresponding to the smallest distance is added to the permutation (tour) at the corresponding, left or right end.

Note that the case of $k = m$, $a_{kL} = a_{Rm}$ is not possible if A is a strong anti-Robinson matrix. Indeed, if k is placed before L, then in the row corresponding to k in the permuted matrix there are two items, a_{kL} and a_{kR}, on one side of the diagonal with equal values because $a_{kR} = a_{Rk} = a_{Rm} = a_{kL}$, which is not allowed in the strong matrix. This means that if $k = m$, then $a_{kL} \neq a_{Rm}$, and hence DENN finds a unique position, left or right, for k.

We first consider the cases when the operation of finding a minimum returns a unique index, i.e., $a_{kL} \neq a_{qL}$ for all $q \neq k$ and $a_{Rm} \neq a_{Rq}$ for all $q \neq m$. According to the structure of a strong anti-Robinson matrix, index k can be added to the permutation either immediately before L or immediately after R. The same two options are possible when placing m. Thus, we consider four cases below.

Case (a). There exists a permutation where index k is before L, and index m is after R; thus, $k \neq m$. The decision rule in DENN will not violate this structure: if $a_{kL} < a_{Rm}$, then k is placed before L and defined as the current left node, otherwise m is placed after R and defined as the current right node.

Case (b). There exists a permutation with indices k and m placed before L. This is possible only if $k = m$. Indeed, if $k \neq m$, then from the definition of k we have $a_{mL} > a_{kL}$, and hence m must be placed before k owing to the structure of the anti-Robinson matrix. On the other hand, from the definition of m, we have $a_{Rm} < a_{Rk}$, which contradicts the definition of the strong anti-Robinson matrix. If $k = m$, then according to the structure of a strong anti-Robinson matrix, we have $a_{Lk} < a_{Rk}$. The decision rule in DENN places k before L as $a_{Lk} < a_{Rm}$, which is precisely its location in the permutation.

Case (c). Indices k and m are after R in a permutation. This is possible only if $k = m$. The proof here is similar to that of Case(b). According to the structure of a strong anti-Robinson matrix, we have $a_{Lk} > a_{Rk}$. The decision rule

in DENN places k after R.

Case (d). Index m is before L and index k is after R in a permutation. We claim that this contradicts the assumption that A is a strong permuted anti-Robinson matrix. Indeed, if m precedes k, then according to the structure of a strong anti-Robinson matrix, we have $a_{Lm} < a_{Rm}$ and $a_{Rk} < a_{Lk}$. From the definition of k, $a_{Lk} \leq a_{Lm}$. By combining these inequalities, we obtain $a_{Rk} < a_{Lk}$, $a_{Lk} \leq a_{Lm}$, $a_{Lm} < a_{Rm}$, and hence $a_{Rk} < a_{Rm}$, which contradicts the definition of m. This proves the claim.

Now consider the case that in a step of DENN two items in a row of the distance matrix correspond to the found minimum. W.l.o.g. we assume that it is the left node L for which two indices k_1 and k_2 have been found such that $a_{Lk_1} = a_{Lk_2} = min_q\{a_{Lq}\}$. If these indices are to be added to the permutation at this step, then one index should be added before L and the other index should be added after R. According to the structure of the anti-Robinson matrix, the minimum for row R at this step also has to be in column k_1 or k_2. Since A is a strong anti-Robinson matrix, we have $a_{Lk_1} < a_{Rk_1}$ and $a_{Rk_2} < a_{Lk_2}$. Since $a_{Lk_1} = a_{Lk_2}$, this yields $a_{Rk_2} < a_{Lk_1}$, and the unique index k_2 is determined in this step of DENN and placed next to R. This means that we can ignore multiple returns in the steps of finding minima.

Note that removing rows and columns from A does not destroy the special structure of the matrix. Thus, in our proof it was enough to consider only the case with two items in a partially constructed permutation. This completes the proof of the lemma. □

As shown above, DENN can start with any start node s. For NN to obtain the permutation for a permuted strong anti-Robinson matrix, it must start with a specific node, as shown in the lemma below.

Lemma 2.2.2 *Given a* permuted *strong anti-Robinson matrix $A = (a_{pq})$, permutation τ constructed in NN permutes the matrix into the strong anti-Robinson matrix if NN starts with node s_1 or node s_2 with $a_{s_1 s_2} = max\{a_{pq}\}$.*

Proof. According to the structures of anti-Robinson matrices, we know that in a strong anti-Robinson matrix, the maximal items are in row 1, column n and row n, column 1 because all items in each row and column are non-decreasing when moving away from the main diagonal in any direction. Considering that the NN heuristic can only add nodes on one side, the first or last item in the permutation should be the index of the row or the column where the maximal item is placed

in the permuted strong anti-Robinson matrix. Therefore, the construction of the permutation can start with node s_1 defined as mentioned in the lemma. In the general case, for the Euclidean matrix this node can be found in $O(nlogn)$ time. For a permuted strong anti-Robinson matrix, a simplified linear time algorithm can be used: start with an arbitrary node p and find s_2: $a_{ps_2} = max_q\{a_{pq}\}$, then find s_1 such that $a_{s_1s_2} = max_q\{a_{qs_2}\}$. Given that s_1 is found correctly, NN finds the permutation by finding the unique available minimum on each step. □

Lemma 2.2.3 *Given a* permuted *strong anti-Robinson matrix $A = (a_{pq})$, permutation τ constructed in GREEDY permutes the matrix into the strong anti-Robinson matrix.*

Proof. The proof of this lemma is similar to that of the previous lemmas. The only new point to consider is how to deal with two sub-paths (chains of nodes) when they are to be connected by a newly found edge. There are four possible ways to combine the two chains in a permutation, one for each possible connecting edge. It is easy to check that, according to the conditions of the strong anti-Robinson matrix, among the four edges that can connect two chains, the unique smallest edge is correctly found in GREEDY. To be more specific, assume that two sub-paths $\langle a, \ldots, b\rangle$ and $\langle c, \ldots, d\rangle$ have already been constructed. There are four ways to connect the two chains: $\langle b, \ldots, a, c, \ldots, d\rangle$, $\langle b, \ldots, a, d, \ldots, c\rangle$, $\langle a, \ldots, b, d, \ldots, c\rangle$, $\langle a, \ldots, b, c, \ldots, d\rangle$. Using the rules in GREEDY, the shortest edge among edges (a, c), (a, d), (b, d), (b, c) is selected to connect the chains. For the permuted strong anti-Robinson matrix, the four edge lengths are unique; therefore, w.l.o.g. we assume that edge (b, d) is the shortest, hence the partial permutation is $\langle a, \ldots, b, d, \ldots, c\rangle$. According to the structures of strong anti-Robinson matrices, $c_{bd} < c_{bc}, c_{bd} < c_{ad}, c_{bd} < c_{ac}$, which is consistent with the GREEDY rule. □

Obviously, not every anti-Robinson matrix is a Kalmanson matrix. Tours that correspond to permutations found in NN, DENN and GREEDY are optimal for the TSP if the underlying distance matrix is both a strong Kalmanson and strong anti-Robinson matrix.

The aim is to slightly modify the above well-known heuristics to obtain an additional theoretical property. Motivated by the Kalmanson matrix, we are now ready to describe the modifications of the heuristics in the next section. Note that the modifications do not change the greedy logic of the heuristics. Moreover, because we will show that the modified heuristics can solve to optimality the TSP

with permuted strong Kalmanson matrices, we call the modified heuristics the Kalmanson heuristics and use the notations K-NN, K-DENN and K-GREEDY.

2.2.2 Kalmanson Heuristics and Theoretical Properties for the TSP

The basic idea of Kalmanson heuristics is simple. Consider the TSP with an $n \times n$ matrix C. We select an arbitrary node, which we refer to as *first* in the tour to be constructed. Then we amend this matrix by subtracting the $first$-related constants from the rows and columns to obtain a distance matrix. Next, the $(n-1) \times (n-1)$ submatrix on the set of indices $\{1, 2, \ldots, n\} \setminus \{first\}$ is considered as the matrix C', and the NN, DENN or GREEDY heuristic is applied to C' to obtain a permutation for the TSP. The solution to the initial TSP is the tour that starts with *first* followed by the permutation.

We start with the K-DENN heuristic. It takes as an input an $n \times n$ distance matrix C and three parameters: $first$, which defines the row and column of the matrix for the transformations, *start*, which defines the node where the search begins, and $\alpha \in [0, 1]$, a real parameter, which will be discussed later. The procedure returns a TSP tour τ. The specific algorithm is given in Algorithm 2.2.4. After the matrix transformation, the tour from *start* is constructed and the nearest node (to the left or right) is added to the partial tour, as in the classical DENN. Note that in the tour construction process, we compare the distances based on the transformed matrix $C^{'}$ instead of the original matrix C.

Algorithm 2.2.4

Algorithm of K-DENN $(n, C, first, start, \alpha)$

$S \leftarrow \{1, 2, 3, \ldots, n\} \setminus \{first, start\}$;$H \leftarrow \langle start \rangle$; $left \leftarrow start$; $right \leftarrow start$;
Transform C *into* C' *with* $c'_{ij} = c_{ij} - \alpha(c_{i,first} + c_{first,j})$, *and focus on*
$(n-1) \times (n-1)$ *submatrix on set of indices* $\{1, \ldots, n\} \setminus \{first\}$;
while $S \neq \emptyset$ ***do***
  $x_1 \leftarrow argmin(c'_{x_1,left})$, $x_2 \leftarrow argmin(c'_{right,x_2})$, *where* $x_1, x_2 \in S$;
  if $c'_{x_1,left} < c'_{right,x_2}$ ***then***
    $H \leftarrow \langle x_1, H \rangle, left \leftarrow x_1$, $S \leftarrow S \setminus \{x_1\}$;
  else
    $H \leftarrow \langle H, x_2 \rangle, right \leftarrow x_2$, $S \leftarrow S \setminus \{x_2\}$;
$\tau \leftarrow \langle H \rangle$; *Insert* $first$ *into* τ *with the minimal increment as a cycle based on* C;
return τ.

The algorithm starts with obtaining the $(n-1) \times (n-1)$ matrix C'. If C is a permuted strong Kalmanson matrix, then C' (obtained with $\alpha = 1$) is a permuted strong anti-Robinson matrix. In this case, DENN as a part of K-DENN can yield a permutation H for C' such that the renumbered matrix satisfies the anti-Robinson conditions. Therefore, the output tour τ is an optimal TSP solution for the permuted strong Kalmanson matrix C.

It can be proved that if the matrix C is a permuted *strong* Kalmanson matrix, then there exists only one shortest TSP tour, and hence τ is the unique permutation that permutes C into the strong Kalmanson matrix C^τ. In other words, an insertion of $first$ into the optimal position would not destroy the property of the permutation because the optimal position is unique. If we aim only at obtaining the permutation for a permuted *strong* Kalmanson matrix, the last step of the optimal insertion of $first$ into H can be replaced by placing $first$ directly in front of H, so that the outcome is $\tau = \langle first, H \rangle$. However, if the initial matrix is an arbitrary distance matrix, the solution obtained is only a heuristic solution for the TSP. In this case, the optimal insertion in the last step seems to be a better operation in the algorithm.

The K-NN heuristic has a similar procedure. It takes the same inputs and returns a TSP tour τ. The specific algorithm is given in Algorithm 2.2.5. In this algorithm, when we apply NN to the transformed matrix C', we choose the left side to extend the tour, although the right side can also be chosen. Note that the

parameter *start* here must be chosen carefully, as shown in Lemma 2.2.2, which will be discussed later. Also, this modified heuristic can obtain the optimal TSP solution for the permuted strong Kalmanson matrix.

Algorithm 2.2.5

Algorithm of K-NN $(n, C, first, start, \alpha)$

$S \leftarrow \{1, 2, 3, \ldots, n\} \setminus \{first, start\}$;
$H \leftarrow \langle start \rangle$; $left \leftarrow start$;
Transform C *into* C' *with* $c'_{ij} = c_{ij} - \alpha(c_{i,first} + c_{first,j})$, *and focus on* $(n-1) \times (n-1)$ *submatrix on set of indices* $\{1, \ldots, n\} \setminus \{first\}$;
while $S \neq \emptyset$ ***do***
  $x \leftarrow argmin\ (c'_{x,left})$, *where* $x \in S$;
  $H \leftarrow \langle x, H \rangle, left \leftarrow x$;
  $S \leftarrow S \setminus \{x\}$;
$\tau \leftarrow \langle H \rangle$; *Insert* $first$ *into* τ *with the minimal increment as a cycle based on* C;
return τ.

The other heuristic motivated by the Kalmanson matrix is K-GREEDY. Its difference from K-DENN and K-NN is that it only has two parameters: $first$ and $\alpha \in [0, 1]$. Similar to K-DENN and K-NN, the initial matrix C is first transformed into the $(n-1) \times (n-1)$ matrix $C' = (c'_{ij})$. All edges (except the $first$-linked edges) are sorted from the smallest to the largest based on the transformed values c'_{ij} and added to form a temporary cycle using the same rules as in GREEDY. The size of the temporary cycle is $n-1$. Finally, we insert $first$ into the cycle with the minimal increment. The specific algorithm is given in Algorithm 2.2.6.

Algorithm 2.2.6

Algorithm of K-GREEDY $(n, C, first, \alpha)$

Transform C *into* C' *with* $c'_{ij} = c_{ij} - \alpha(c_{i,first} + c_{first,j})$*, and focus on* $(n-1) \times (n-1)$ *submatrix on set of indices* $\{1, \ldots, n\} \setminus \{first\}$*;*
Sort all c'_{ij} *from the smallest to largest to form a whole edge list* WE *;*
$k \leftarrow 0$*;* $degree_i \leftarrow 0$ *for* $i = \{1, 2, \ldots, n\}$*;*
while $k < n - 2$ ***do***
  Select the shortest edge $(i, j) \in WE$*;*
  if $degree_i < 2$ ***and*** $degree_j < 2$ ***and*** *a cycle will not be formed with* (i, j) ***then***
    Add (i, j) *to* τ*;*
    $degree_i \leftarrow degree_i + 1$*;* $degree_j \leftarrow degree_j + 1$*;* $k \leftarrow k + 1$*;*
  $WE \leftarrow WE \setminus \{(i, j)\}$*;*
Add (i, j) *to* τ*, where* $degree_i = degree_j = 1$*;*
Insert $first$ *into* τ *with the minimal increment as a cycle based on* C*;*
return τ*.*

As described, the Kalmanson heuristics: K-NN, K-DENN and K-GREEDY can obtain the optimal TSP tour for permuted strong Kalmanson matrices. The theorem is given below.

Theorem 2.2.7 *K-NN, K-DENN and K-GREEDY (obtained with* $\alpha = 1$*) solve the TSP with a permuted strong Kalmanson matrix to optimality.*

Note that when Kalmanson heuristics are used for a permuted strong Kalmanson matrix, any node can be chosen as the parameters *start* and $first$ for K-DENN; for K-NN, *start* should be chosen carefully as the node on the longest edge of the transformed matrix C' (see Lemma 2.2.2), but $first$ can be randomly chosen; for K-GREEDY, *start* is not required, and any node can be chosen as $first$.

Now, we discuss the reasons for introducing parameter α. As discussed, a Kalmanson matrix can be transformed into an anti-Robinson matrix using $c'_{ij} = c_{ij} - (c_{i,first} + c_{first,j})$. Therefore, if $\alpha = 1$, then the TSP with a permuted strong Kalmanson matrix can be solved to optimality by Kalmanson heuristics. If $\alpha = 0$, the Kalmanson heuristics can still obtain the permutation for a permuted strong anti-Robinson matrix, as the original matrix will not be changed after the

transformation and the classical heuristics (NN, DENN and GREEDY) as a part of the Kalmanson heuristics can yield a permutation.

We use a case where all nodes are located on a straight line to demonstrate the application of $\alpha < 1$. In this case, the corresponding distance matrix C is not a permuted strong Kalmanson matrix; thus, transforming C into C' with $\alpha = 1$ does not help in obtaining the correct sequence of the nodes. However, C itself is a permuted strong anti-Robinson matrix. Therefore, setting parameter α to 0 will permute C into a strong anti-Robinson matrix, thus obtaining the correct sequence of the nodes in this case. Actually, any $\alpha < 1$ (not necessarily $\alpha = 0$) will be sufficient to solve the case to optimality when $first$ and $start$ are carefully chosen. We take K-DENN as an example.

Proposition 2.2.8 *If $first$ and $start$ are the two endpoints on a straight line, then K-DENN with coefficient $0 < \alpha < 1$ will obtain the correct sequence of nodes on the line for the TSP.*

Proof. W.l.o.g. define $first(f)$ and $start(s)$ as the rightmost and leftmost nodes on the line respectively. Construct the tour from s. k is chosen from the remaining nodes to minimise $c'_{sk} = c_{sk} - \alpha(c_{sf} + c_{fk}) = (1 + \alpha)c_{sk} - 2\alpha c_{sf}$. Because c_{sf} is constant, k should be the node closest to s, i.e., the leftmost node from the remaining nodes. Then place k on the left or right side of s; here we assume the right side. Now, the partial tour is $\langle L, R\rangle$, where $L = s$, $R = k$. Next, choose nodes a and b to minimise c'_{La} and c'_{Rb}, where $c'_{La} = (1+\alpha)c_{La} - 2\alpha c_{Lf}$, $c'_{Rb} = (1+\alpha)c_{Rb} - 2\alpha c_{Rf}$. Therefore, a and b are both the leftmost nodes from the remaining nodes, and we assume $a = b = m$. Then c'_{Lm} and c'_{Rm} are compared to decide which side to add, where $c'_{Lm} = (1+\alpha)c_{Lm} - 2\alpha(c_{Lm} + c_{mf}) = (1-\alpha)c_{Lm} - 2\alpha c_{fm}$, $c'_{Rm} = (1-\alpha)c_{Rm} - 2\alpha c_{fm}$. When $\alpha = 1$, $c'_{Lm} = c'_{Rm}$; thus, m can be added to either side. If m is added to the left side of the partial tour, then the constructed permutation is not optimal. This demonstrates why K-DENN with coefficient $\alpha = 1$ is invalid. However, when $0 < \alpha < 1$, m can only be added to the right side because $c'_{Rm} < c'_{Lm}$. By parity of reasoning, R is always updated with the leftmost node from the remaining nodes until only $first$ is left. Then we insert $first$ at the position with the minimum increment in length as a cycle. If $first$ is placed at the front or the back of the partial tour, then the increment is $\Delta_1 = c_{Rf} + c_{fL} - c_{RL} = 2c_{Rf}$. If it is inserted in the middle (we assume between i and j), then $\Delta_2 = c_{if} + c_{fj} - c_{ij} = 2c_{fj}$. Therefore, $first$ should be placed at the front or the back because $\Delta_1 \leq \Delta_2$. Now the tour $\langle L, f\rangle$ is constructed from the nodes numbered along the line, which is the

correct permutation for the permuted strong anti-Robinson matrix and also the optimal tour. □

Having derived some theoretical results for the Kalmanson heuristics, we next discuss their empirical investigation. When the Kalmanson heuristics are applied to arbitrary distance matrices to obtain heuristic solutions, one can try α from the range $[0, 1]$. Preliminary experiments suggest that each of the Kalmanson heuristics has its best α on the set of instances tested. This will be illustrated in the next section.

The Kalmanson heuristics have another important feature for the TSP: the number of crossings is smaller than that for NN, DENN and GREEDY. The previous diamond instance shown in Figure 2.1 also illustrates the *crossing* feature of the Kalmanson heuristics. The classical heuristics will always include edge $(2, 4)$ in the TSP tour, independently of *start*. This means that the tour will always have one crossing. In contrast, the Kalmanson heuristics will always find the optimal tour, which is $\langle 1, 2, 3, 4, 1\rangle$ (or its inverse and/or cyclic shift equivalents) and has no crossings. We take K-DENN as an example, with node 1 as $first$. After transforming matrix D into matrix $D' = (d'_{ij})$ with zeros in the first row and column, we have $d'_{23} = d'_{32} = d'_{34} = d'_{43} = -24$, $d'_{24} = d'_{42} = -16$. If we choose, for instance, 3 as the *start*, then the tour will grow as $H = \langle 3\rangle$, $H = \langle 3, 2\rangle$, and $H = \langle 4, 3, 2\rangle$. To insert 1 into the best position, we compare $d_{21} + d_{14} - d_{24} = 13 + 13 - 10 = 16$, $d_{41} + d_{13} - d_{43} = 24$ and $d_{31} + d_{12} - d_{32} = 24$. The best insertion will lead to the tour $\langle 1, 4, 3, 2, 1\rangle$, which also defines the sequence of the points on their convex hull. More comparisons are illustrated in the experiments in Section 2.3.

2.3 Empirical Investigation of Kalmanson Heuristics

To demonstrate the superiority of our Kalmanson heuristics to the classical counterparts, we conducted experiments on the TSP DIMACS Challenge test suite which has known optimal solutions [90]. The test suite contains six benchmark instance sets: two random uniform Euclidean instance sets E1K and E3K, two random clustered Euclidean instance sets C1K and C3K, and two TSPLIB instance sets TSPLIB1 and TSPLIB2. E1K, C1K, E3K and C3K contain 10, 10, 5 and 5 instances respectively. Here, $1K$ and $3K$ represent the sizes 1000 and 3162 respectively. TSPLIB1 contains 22 single instances, which have sizes between 1000 and 3795; TSPLIB2 contains 43 instances with sizes between 100 and 783. The size

of TSPLIB instances is determined by the numerical postfix. Detailed information of the test instances is given in Appendix A. The algorithms are coded in C++ and executed on an Intel Core i5-7500 3.40 GHz processor with 8.0 GB RAM.

Because we retain the simplicity of the classical heuristics, the difference in computational time between the K-versions and the original versions is negligible. (Note that the time for the instances with 1000 points is only a fraction of a second.) Therefore, we only compare the tour quality when comparing the Kalmanson heuristics with the original counterparts. For each instance, the tour length is recorded as v, the optimal objective value is recorded as v_{opt} and the Held-Karp (HK) lower bound is denoted as v_{hk}. Note that the HK lower bound v_{hk} is a linear programming relaxation of the standard integer programming for the TSP, which is a good surrogate for the optimal solution value [91]. Two measures are used to calculate the gap for each instance, which we use as an indicator of the tour quality: the percentage excess over the optimal solution, $opt\% = \frac{v-v_{opt}}{v_{opt}} \times 100\%$, and the percentage excess over the HK lower bound: $hk\% = \frac{v-v_{hk}}{v_{hk}} \times 100\%$. For each instance set, the average values of $opt\%$ and $hk\%$, respectively denoted as $OPT\%$ and $HK\%$, are calculated.

Before we compare different algorithms, we perform the preliminary experiments outlined below.

First, the parameters $first$ and $start$ need to be determined for Kalmanson heuristics. Actually, one can use an $O(n^4)$ version to explore all possible pairs of $first$ and $start$ and find the best tour constructed. However, to increase the efficiency, we need to find a better way to generate $start$ and $first$.

The theoretical results suggest a suitable approach to finding $first$ and $start$. As discussed, for K-NN to be able to solve a permuted strong Kalmanson matrix to optimality, $start$ must be a node on the longest edge of the transformed matrix $C' = (c'_{ij})$. Therefore, using the endpoints of long edges as $start$ and $first$ is expected to produce good parameters. In what follows, two different procedures are used to find P long edges and thus produce P pairs of the two parameters $first$ and $start$.

We begin with K-DENN. The First Procedure is to find the P longest edges among all edges based on the original matrix C, and sort the P edges in a list E from the longest to the shortest. Each edge corresponds to each pair of parameters $(first, start)$: the two endpoints of the edge are used as $first$ and $start$. In the Second Procedure, the farthest node from $first$ is always selected as $start$ on the basis of C, and P pairs are generated as follows: the depot (node 1) is chosen as

$first$ of the first pair, and the corresponding *start* is determined; then the current *start* is used as the next $first$ until P pairs are obtained. The P pairs are also stored in a list E. The First Procedure and the Second Procedure have complexity of $O(n^2P)$ and $O(nP)$ and are given in Procedure 2.3.1 and Procedure 2.3.2 respectively.

Procedure 2.3.1

First Procedure (n, C, P)

Size of edge list $p \leftarrow 0$; $E \leftarrow \emptyset$;
A whole edge list $WE \leftarrow \{(i,j)\}$ *for* $i = 1, 2, \ldots, n-1, j = i+1, \ldots, n$;
while $WE \neq \emptyset$ ***do***
    Keep the pairs (i,j) *in* E *sorted from the largest to smallest based on* c_{ij};
    if $p < P$ ***then***
        Add the pair (i,j) *to the sorted position of* E;
        $p \leftarrow p + 1$;
    else
        Insert the pair (i,j) *to the sorted position of* E;
        Remove the last pair in E;
    $WE \leftarrow WE \setminus \{(i,j)\}$;
return E.

Procedure 2.3.2

Second Procedure (n, C, P)

$S \leftarrow \{1, 2, 3, \ldots, n\}$; $E \leftarrow \emptyset$;
$p \leftarrow 0$; $first \leftarrow 1$;
while $p \leq P - 1$ ***do***
    $start \leftarrow argmax(c_{first,start})$, *where* $start \in S$; *add (first,start) to* E;
    $first \leftarrow start$;
    $S \leftarrow S \setminus \{start\}$;
return E.

For K-NN, we use similar ideas to those in the two procedures for K-DENN to generate $first$ based on C, but we generate *start* from the transformed matrix C'. Specifically, in the First Procedure, the P longest edges (i,j) are found and i

is used as $first$. Then one focuses on the $(n-1) \times (n-1)$ transformed matrix C', finds the P longest edges (i', j') and uses j' as $start$. A similar method is applied to the Second Procedure. The time complexity of the First Procedure and the Second Procedure is still $O(n^2P)$ and $O(nP)$ respectively.

For K-GREEDY, $start$ is not needed, but similar procedures to above are used to generate P pairs, where only $first$ is recorded. The two procedures are investigated below to find the best parameter pair for Kalmanson heuristics.

First, the effects of the P value on the performance of a multi-start are tested for Kalmanson heuristics. A multi-start here means that for each instance, P parameter pairs are used to obtain P solutions, and the best solution is recorded as the multi-start solution τ with objective value v. We perform tests on benchmark instances using two procedures with different P values and calculate the average gap $OPT\%$ for each instance set. The results in Figures 2.2 and 2.3 are for instance sets C1K and E1K, respectively, using K-DENN as an example.

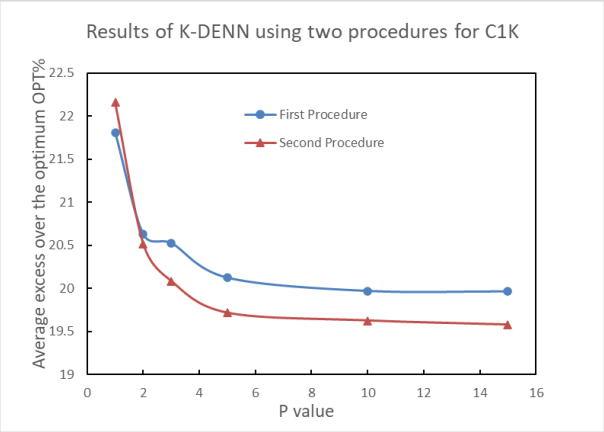


Figure 2.2: Comparison of two procedures on K-DENN: average gap $OPT\%$ with changing P for C1K

These figures suggest two findings. The first one is that the Second Procedure performs better than the First Procedure in terms of average gap. Considering that the Second Procedure uses less time than the First Procedure, we hereafter use the Second Procedure to generate parameter pairs. The other finding is that as P increases, the improvement of solutions when $P > 2$ is less than that when $P \leq 2$. Therefore, we only focus on the first two pairs produced by the Second Procedure in the further experiments.

In the next step, we explore whether the first pair or the second pair from the Second Procedure is better. Before discussing the experiments, we consider

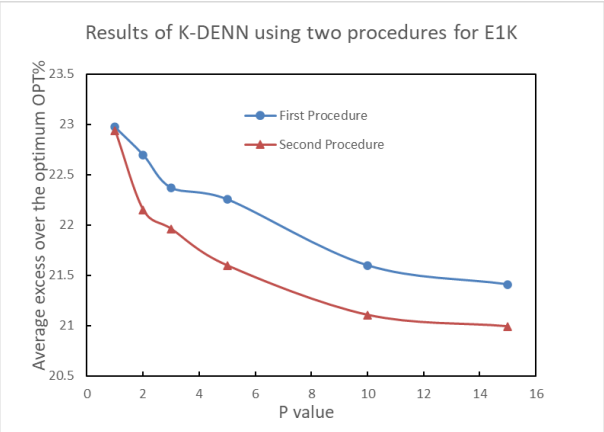


Figure 2.3: Comparison of two procedures on K-DENN: average gap $OPT\%$ with changing P for E1K

the theoretical implications. As shown in Lemma 2.2.2, for K-NN to be able to solve a permuted strong Kalmanson matrix to optimality, *start* must be a node on the longest edge of the transformed matrix $C' = (c'_{ij})$. Considering the property of the anti-Robinson matrix, *start* can be found using a linear time algorithm by starting with an arbitrary node k and finding s_1 such that $c'_{ks_1} = max_q\{c'_{kq}\}$, then finding s_2 such that $c'_{s_1s_2} = max_q\{c'_{s_1q}\}$. Using the linear algorithm is equivalent to obtaining the second pair from the Second Procedure. This suggests that if we use the second pair from the Second Procedure, K-NN can solve the permuted strong Kalmanson matrices to optimality. The theoretical result indicates that the second pair from the Second Procedure may be the better option for Kalmanson heuristics.

We run the experiments with a single start (single pair) and record the results $OPT\%$ of the first and second pairs separately in Table 2.1. The results show that although there is not a large difference between the two pairs, the second pair performs better (except on TSPLIB2). This computational result is consistent with the above theoretical discussion; thus, we choose the second pair from the Second Procedure for the Kalmanson heuristics in the later experiments.

Then the effects of the parameter α on the solution quality are investigated for the three Kalmanson heuristics. To select the best α, we first observe the performances on self-generated instances with different values of α before applying them to the benchmark instances.

We generate 200 instances including 50 random uniform Euclidean instances and 50 random clustered Euclidean instances for problem sizes of 1000 and 3162,

Table 2.1: Comparison of average gap $OPT\%$ between two pairs from the Second Procedure using K-DENN

	first pair	second pair
C1K	22.16%	20.98%
E1K	22.94%	22.47%
C3K	27.24%	27.19%
E3K	25.35%	25.28%
TSPLIB1	24.49%	24.34%
TSPLIB2	15.95%	16.01%

which are consistent with the benchmark instances. Thus, there are four instance sets in total: R-C1K, R-E1K, R-C3K and R-E3K. The generation process is that used to generate the Challenge test suite in [90] and is described below.

Random uniform Euclidean instances The city points have two integer coordinates chosen randomly from the uniform distribution $(0, 1000000]$. Distances are Euclidean distances and are rounded to the nearest integer.

Random clustered Euclidean instances We choose 10 and 30 cluster centres whose coordinates are generated randomly from the uniform distribution: $(0, 1000000]$ for the problems of size $n = 1000$ and 3162 respectively. We first assign a random cluster centre to each city node. Then for each city node, two coordinates are generated from the standard normal distribution, multiplied by $\frac{1000000}{\sqrt{N}}$, rounded, and added to the corresponding coordinate of its centre. Distances are also Euclidean distances rounded to the nearest integer.

For each of the three Kalmanson heuristics, we change α from 1 to 0, i.e., $1, 0.9, 0.8, \ldots, 0.1, 0$, and use the second pair $(first, start)$ from the Second Procedure to obtain solutions. For each instance, solutions with different α are compared and the best solution is used as the comparison base, from which we calculate the percentages. For each α, the average percentage is calculated within each instance set. Figures 2.4 - 2.6 show the average gap for each group with changing α using K-DENN, K-NN and K-GREEDY respectively.

These figures show that for each instance set, whichever Kalmanson heuristic is used (except R-C1K using K-GREEDY), there are similar tendencies as α changes, with each curve having a V shape. Although the valley values and the sinuosity vary among the Kalmanson heuristics, the tendencies are similar when using the same Kalmanson heuristic among different instance groups. Therefore, for each Kalmanson heuristic, we can find the 'best' α that can be applied to dif-

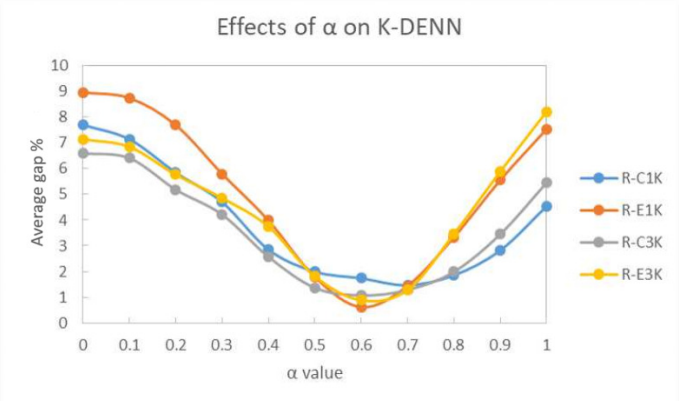


Figure 2.4: Effects of α on average gap for K-DENN on self-generated instances

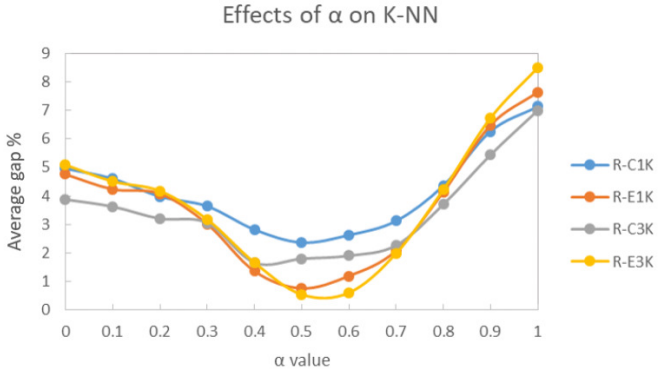


Figure 2.5: Effects of α on average gap for K-NN on self-generated instances

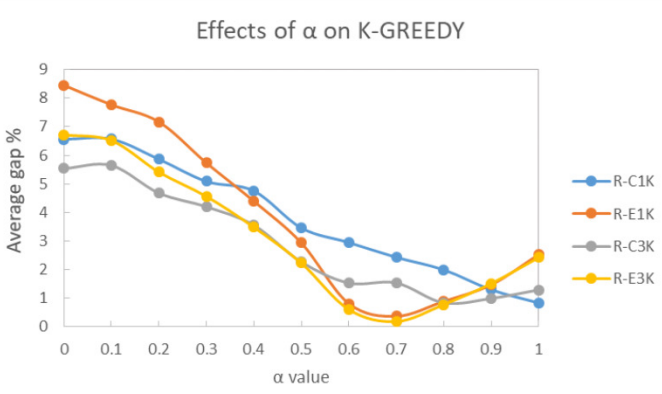


Figure 2.6: Effects of α on average gap for K-GREEDY on self-generated instances

ferent instance sets ideally. Taking K-DENN as an example, when $\alpha = 0.6$, the average gap is lowest for groups R-E1K, R-C3K and R-E3K and relatively low for group R-C1K; thus, generally 0.6 is the best option. The best α value is found to be $\alpha = 0.6, 0.5$ and 0.7 for K-DENN, K-NN and K-GREEDY respectively.

It is worth mentioning that the 'best' α chosen above is only at a general level. If a new instance has a clustered node distribution with a size of 1000, which is similar to the attribute in R-C1K, a better option for α will be 0.7, 0.5 and 1.0 when using K-DENN, K-NN and K-GREEDY respectively. One can also investigate the effects of α on a single instance and choose the 'best' α to obtain a better solution. In this process, finer intervals, e.g., $\alpha = 0.01, 0.02, \ldots, 0.99, 1$, can also be explored, but we need to consider the trade-off between computational time and solution quality, which is the most common issue in TSP related problems.

Note that there are two different operations in the last step in Kalmanson heuristics: optimal insertion of $first$ into the partial tour or placing $first$ directly in front of the partial tour. So far, we have always used the optimal insertion of $first$, whose importance can be demonstrated by comparison with directly placing $first$. Taking K-NN with its best value of $\alpha = 0.5$ as an example, Figure 2.7 suggests a marked improvement with optimal insertion. Considering the minor difference in computational time, the optimal insertion is used in the following experiments.

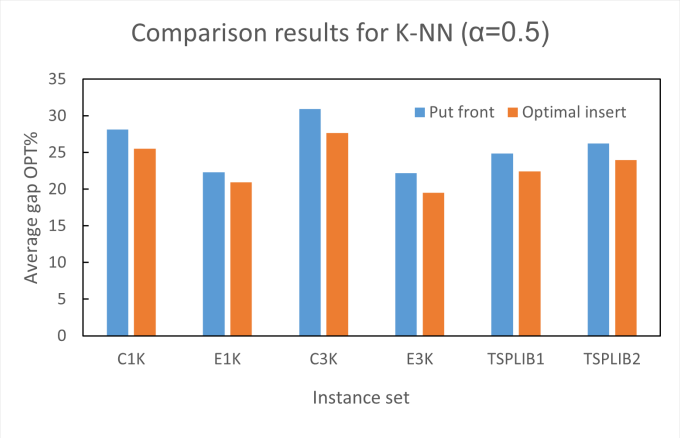


Figure 2.7: Comparison of results of optimal insertion of $first$ and placing $first$ in front for K-NN with $\alpha = 0.5$

Based on the preliminary experiments, we can now demonstrate the performance of the Kalmanson heuristics and compare them with their classical counter-

parts using benchmark instances. We use the second pair of $(first, start)$ from the Second Procedure, apply the best α values and execute the optimal insertion for each Kalmanson heuristic. To ensure a reasonable and fair comparison, the Second Procedure is also used to generate *start* for the classical heuristics. The results of the comparison are summarised in Figures 2.8 - 2.10. To describe the variation within each instance set, we use box plots, which graphically depict groups of numerical data through their quartiles. The minimum and maximum values represent the best and worst performance respectively. The mean values are marked as 'x' to reflect the average performance.

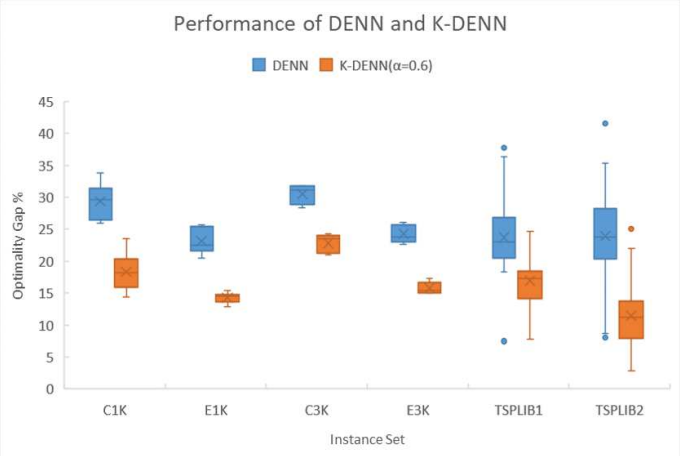


Figure 2.8: Comparison of DENN and K-DENN($\alpha = 0.6$) on benchmark instances

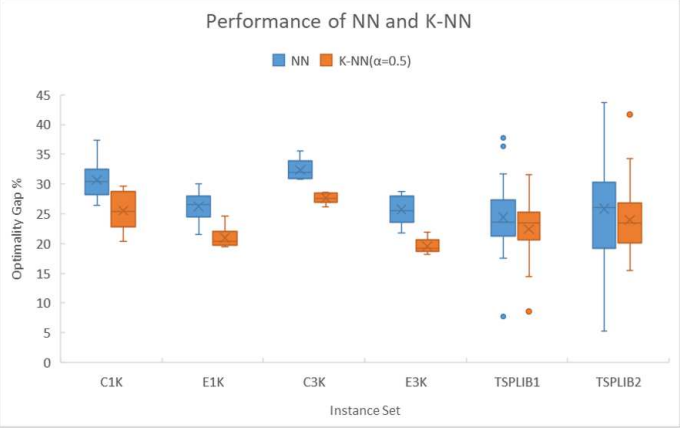


Figure 2.9: Comparison of NN and K-NN($\alpha = 0.5$) on benchmark instances

These figures indicate that the average performances of the three Kalmanson

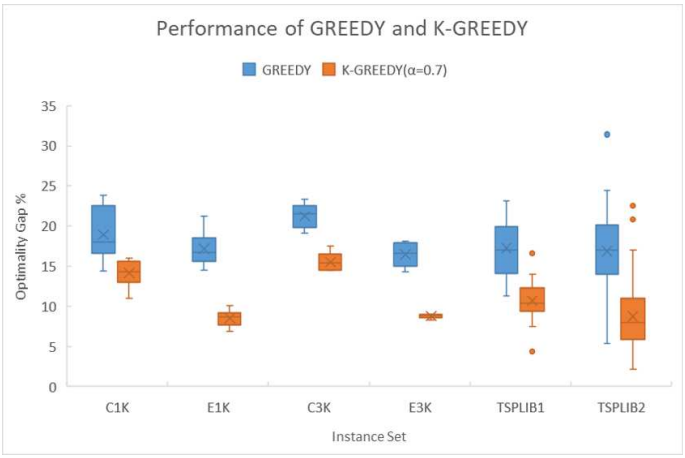


Figure 2.10: Comparison of GREEDY and K-GREEDY($\alpha = 0.7$) on benchmark instances

heuristics with the best α are all superior to the classical counterparts, especially for K-DENN and K-GREEDY. Table 2.2 also displays the same results. When we focus on the best and worst performance, the Kalmanson versions are also superior to the classical versions (except TSPLIB instances using NN). In addition, the ranges of boxes for the Kalmanson heuristics are narrower, which suggests that the performance of the Kalmanson heuristics is more robust.

Table 2.2: Comparison of results of average excess over optimal solutions: $OPT\%$ using Kalmanson heuristics (with best α) and classical versions

	NN	K-NN	DENN	K-DENN	GREEDY	K-GREEDY
C1K	30.69	25.51	29.36	18.33	18.90	14.15
E1K	26.25	20.94	23.13	14.25	17.16	8.50
C3K	32.34	27.65	30.48	22.82	21.24	15.49
E3K	25.71	19.53	24.26	15.78	16.47	8.76
TSPLIB1	24.40	22.43	23.70	16.97	17.24	10.65
TSPLIB2	25.83	23.95	23.91	11.55	16.83	8.74

To ensure consistency with the results in [90], the average excess over the HK lower bounds $HK\%$ is also calculated. The results are given in Table 2.3.

Table 2.4 provides the number of instances improved using the Kalmanson heuristics (with best α) compared with the classical counterparts. We found that the Kalmanson heuristics can improve the performance for most instances, especially for the first four instance sets. K-DENN and K-GREEDY perform much

Table 2.3: Comparison of results of average excess over HK lower bound: $HK\%$ using Kalmanson heuristics (with best α) and classical versions

	NN	K-NN	DENN	K-DENN	GREEDY	K-GREEDY
C1K	31.40	26.18	30.06	18.96	19.54	14.76
E1K	27.18	21.83	24.04	15.09	18.02	9.30
C3K	33.15	28.43	31.28	23.57	21.99	16.20
E3K	26.59	20.37	25.14	16.60	17.29	9.53
TSPLIB1	25.69	23.72	24.98	18.16	18.46	11.81

better than the counterparts.

Table 2.4: Number of instances improved by Kalmanson heuristics (with best α) compared with classical counterparts

	Number of instances	K-DENN VS DENN	K-NN VS NN	K-GREEDY VS GREEDY
C1K	10	10	9	10
E1K	10	10	10	10
C3K	5	5	5	5
E3K	5	5	5	5
TSPLIB1	22	19	12	22
TSPLIB2	43	39	25	38
Total	95	88	66	90

As discussed in Section 2.2.2, the Kalmanson heuristics have an important feature for the TSP of having fewer crossings than the classical heuristics. The TSPLIB instance kroB100 of size 100 in [90] is used to illustrate this feature. It is found that the tour using DENN in Figure 2.11 has many intersections, while the tour using K-DENN in Figure 2.12 has no intersections. Table 2.5 presents the average number of intersections for each instance set using different heuristics. Among all 95 benchmark instances, compared with the classical heuristics, the Kalmanson versions decrease the number of crossings for 81, 92 and 94 instances with K-NN, K-DENN and K-GREEDY respectively.

2.4 Conclusions

We have considered three classical heuristics for the TSP: NN, DENN and GREEDY, and have proved that they have the theoretical property of obtaining the permutation for the permuted strong anti-Robinson matrices such that the renumbered matrices satisfy the anti-Robinson conditions.

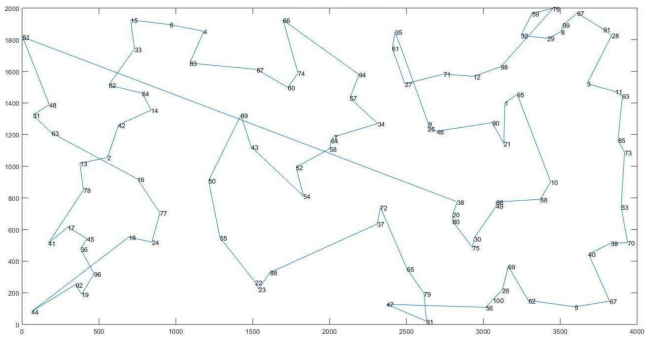


Figure 2.11: Demonstration of crossing feature: the tour of instance kroB100 using DENN

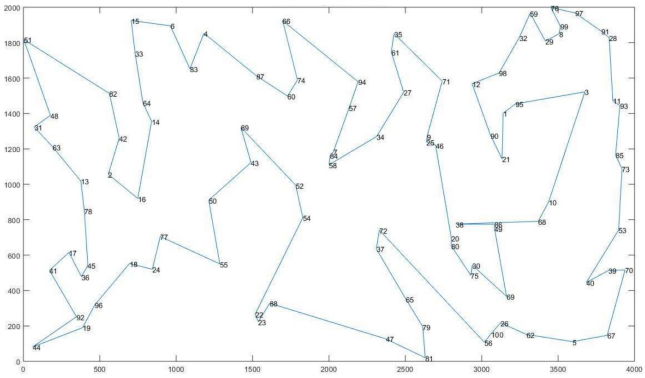


Figure 2.12: Demonstration of crossing feature: the tour of instance kroB100 using K-DENN

Table 2.5: Comparison of average number of crossings on benchmark instances using Kalmanson heuristics and classical counterparts

	DENN	K-DENN	NN	K-NN	GREEDY	K-GREEDY
C1K	102	33	112	61	62	21
E1K	92	18	104	53	67	9
C3K	321	122	333	198	185	58
E3K	283	58	302	142	203	21
TSPLIB1	178	49	183	90	109	25
TSPLIB2	30	6	30	19	20	5

Inspired by the knowledge that the Kalmanson matrix can be transformed into the anti-Robinson matrix, we have proposed minor amendments to the three classical heuristics. The amended versions have the additional theoretical property of obtaining an optimal TSP solution for a permuted strong Kalmanson matrix. The incorporation of additional features from the Kalmanson matrix into the classical heuristics has enriched the nature of the heuristics.

Given an arbitrary distance matrix, Kalmanson algorithms can be used to generate heuristic solutions. The empirical investigation on the benchmark instances indicates that the Kalmanson heuristics, especially K-DENN and K-GREEDY, perform better than the original counterparts.

As future research, we will carry out further theoretical work to determine how far a given distance matrix is from a Kalmanson matrix and/or an anti-Robinson matrix. If we can quantify this distance, we can further quantify the approximation ratio of the Kalmanson heuristics. We will also investigate other heuristics using the knowledge of solvable cases for the TSP.

Chapter 3

Simple Heuristics for the CTSPP

3.1 Introduction and Related Works

The literature suggests that the CTSPP is computationally harder to solve to optimality or even approximately than the TSP [108]. However, the research on heuristics [18] for the CTSPP is deficient when compared with that for the TSP [133, 139, 105, 25, 117]. For the CTSPP, the GRASP heuristic [62] is commonly used to construct an initial solution. Regarding tour improvement heuristics, the classical simple heuristics for the TSP such as 2-opt and 3-opt can be applied to the CTSPP with a cumulative objective function.

One contribution of this chapter is that it enriches the research on heuristics for the CTSPP by proposing three tour improvement heuristics based on dynamic programming, the pyramidal heuristic, chains heuristic and sliding window heuristic, and two tour construction heuristics, an improved version of GRASP (IGRASP) and random insertion heuristic (RIH).

To demonstrate the potential of the proposed heuristics, they will be compared with classical simple algorithms. The proposed algorithms use IGRASP or RIH to construct the tour and the pyramidal heuristic, chains heuristic, or sliding window heuristic to improve the tour. The classical simple algorithms adopt GRASP to generate the initial solutions and classical simple heuristics to improve the solutions. The proposed heuristics are compared with simple heuristics because the proposed heuristics also have simple procedures. To determine the best classical simple heuristic to use for comparison in the tour improvement phase, it is first

necessary to understand the power of different simple heuristics.

In the literature, the analyses and comparisons of classical simple heuristics have been neglected for the CTSPP, especially compared with the extensive analysis of the heuristics for the TSP. Johnson and McGeoch [90] provided detailed analyses and comparisons among various groups of heuristics with regard to scalability, robustness and the trade-off between the running time and solution quality for the symmetric and asymmetric TSP, and they established a mechanism for further comparability. A DIMACS Implementation Challenge has been organized to update the state of the art for the TSP by reporting the solutions on test suites for different algorithms [90]. Other analyses and comparisons of heuristics for the TSP have been reported by Bentley [17], Reinelt [129], Punnen and Kabadi [125] and Punnen et al. [126].

However, for the CTSPP, only one relevant study has been reported, in which 20 instances of the same size were tested [113]. To address this lack of testing, we therefore first analyse and compare the classical simple heuristics for the CTSPP by performing extensive computational experiments. This extensive comparison is another contribution that this chapter makes to the empirical research on the CTSPP.

Metaheuristics for the CTSPP have also been explored. Salehipour et al. [134], Silva et al. [137] and Mladenović et al. [113] developed metaheuristics that use GRASP for tour construction and the VND or VNS for tour improvement. The VND and VNS [112] are advanced local search procedures that combine several simple heuristics to systematically explore different neighbourhood structures (usually five or six). The subordinate heuristics contain the well-known 2-opt, 3-opt, 1-insertion, 2-insertion and 3-insertion heuristics, and each heuristic corresponds to one neighbourhood structure. Essentially, when one local optimum relative to a certain neighbourhood is found, the metaheuristic switches to another neighbourhood. In this process, the order of neighbourhoods may be important. Salehipour et al. [134] adopted a deterministic order in which neighbourhoods are sorted in order of increasing size, i.e., a large neighbourhood can only be searched when the current solution is a local optimum for all smaller neighbourhoods. Silva et al. [137] adopted a random order of neighbourhoods in the VND. There may still be scope to improve the metaheuristics if neighbourhoods are searched in a more efficient order.

Comparing the classical simple heuristics can not only help determine the best heuristic to adopt for comparison with the proposed heuristics, but also in-

crease understanding of the effectiveness of each simple neighbourhood search, thus providing guidance on improving the selection and combination of simple heuristics in the future study of metaheuristics for the CTSPP.

This chapter is structured as follows.

Section 3.2 formulates the problem and analyses the differences between the CTSPP and the Path TSP. Then two heuristics for tour construction are proposed: IGRASP and RIH. Furthermore, seven classical tour improvement heuristics for the CTSPP are demonstrated and analysed in terms of their complexity.

In Section 3.3, we propose three heuristics based on dynamic programming for tour improvement: the pyramidal heuristic, chains heuristic and sliding window heuristic. These heuristics have simple procedures and are motivated by solvable cases. The pyramidal heuristic is motivated by the solvable pyramidal neighbourhood. The chains heuristic and sliding window heuristic are motivated by solvable small-size problems.

In Section 3.4, extensive computational experiments are reported. Firstly, we compare the effectiveness of the seven classical tour improvement heuristics when they are combined with three tour construction heuristics, GRASP, IGRASP and RIH, on self-generated general instances of the CTSPP. On the basis of the experimental results, we select GRASP-2-opt as the comparison base because 2-opt obtains the best solutions among the classical simple tour improvement heuristics. The results also suggest that IGRASP and RIH perform better than the commonly used GRASP when they are combined with classical simple heuristics.

Secondly, we compare the proposed algorithms with GRASP-2-opt on benchmark instances. When using a single start, the proposed algorithms obtain solutions of higher quality but require a longer running time. When the experimental time is controlled, all the proposed algorithms outperform GRASP-2-opt for size $n \geq 200$. Moreover, RIH with one setting of the sliding window heuristic outperforms GRASP-2-opt for all experimental problem sizes. This suggests that the proposed algorithms are promising for the CTSPP. A further experiment is performed on the sliding window heuristic to investigate its performance as the number of starts increases.

Thirdly, experiments are conducted on the solvable specially structured cases of edge-unweighted trees. Minieka [111] stated that the depth-first route is optimal for the CTSPP on unweighted trees. We conduct experiments on this special instance set to demonstrate the high quality of the proposed algorithms. Although the proposed algorithms cannot solve the special cases to optimality,

high-quality solutions are obtained.

Fourthly, we compare the proposed algorithms with GRASP-2-opt on the specially structured cases of edge-weighted trees. It is known that the CTSPP on edge-weighted trees is NP-hard [138]. Considering that the proposed dynamic programming heuristics are based on solvability and that weighted trees are related to solvable unweighted trees, the proposed algorithms are expected to deliver a good performance on the special case-weighted trees. The results reveal that the performance of the proposed algorithms is better than that of GRASP-2-opt across all experimental sizes and suggest that the larger the instance size, the greater the superiority of the algorithms compared with GRASP-2-opt.

Section 3.5 concludes this chapter by summarising the findings.

3.2 Problem Formulation and Simple Heuristics

In this section, we introduce the relevant mathematical definitions and notations for the CTSPP. We also propose tour construction heuristics and analyse the classical simple tour improvement heuristics.

3.2.1 Problem Formulation

The objective of the CTSPP is to find a Hamiltonian path starting from the depot that minimises the sum of arrival times at all customers. Given an $n \times n$ distance matrix $C = (c_{ij})$, where c_{ij} denotes the travelling time between customers i and j, a tour is defined as $\tau = \langle \tau_1, \tau_2, \tau_3, \ldots, \tau_{n-1}, \tau_n \rangle$, where τ_i is the i^{th} node in the solution. Note that τ_1 is the depot. $l(\tau_i)$ is defined as the arrival time (latency) of the i^{th} vertex, which can be written as:

$$l(\tau_i) = \sum_{j=1}^{j=i-1} c_{\tau_j \tau_{j+1}} \tag{3.1}$$

Note that for the CTSPP, the time travelling between the last customer and the depot τ_1 is not included in the total cost, as the vehicle has no time pressure after serving the last customer in a customer-centric service system. Thus, the objective value, the sum of arrival times at all vertices (except the depot), can be written as:

$$c(\tau) = \sum_{i=2}^{i=n} l(\tau_i) = \sum_{i=2}^{i=n} \sum_{j=1}^{j=i-1} c_{\tau_j \tau_{j+1}} \tag{3.2}$$

This is equivalent to the following more computationally efficient equation:

$$c(\tau) = \sum_{i=1}^{i=n-1} (n-i) \times c_{\tau_i \tau_{i+1}} \tag{3.3}$$

We compare the CTSPP with the Path TSP, which is a special version of the TSP. For the Path TSP, the objective is the same as that of the TSP, which is to find an optimal tour to minimise the total travelling time for one vehicle to visit all given customers exactly once, but the vehicle does not need to return to the start point. The different objectives for the Path TSP and the CTSPP result in different optimal solutions given the same following matrix A.

$$A = \begin{pmatrix} 0 & 8 & 2 & 11 \\ 8 & 0 & 5 & 15 \\ 2 & 5 & 0 & 1 \\ 11 & 15 & 1 & 0 \end{pmatrix} \tag{3.4}$$

For the Path TSP, the optimal solution is $\langle 1, 2, 3, 4 \rangle$ with a value of 14, while the value of this tour for the CTSPP is 35. For the CTSPP, the optimal solution is $\langle 1, 3, 4, 2 \rangle$ with a value of 23, while the value of this tour for the Path TSP is 18. The cumulative characteristic of the CTSPP means that edge values are added different numbers of times depending on the positions of the edges, while for the Path TSP, all edge values are only added once. Equation (3.3) also indicates the cumulative characteristic of the CTSPP: the coefficients of the edge values differ for the CTSPP, while they are always 1 for the Path TSP.

3.2.2 Simple Tour Construction Heuristics

Tour construction heuristics construct a tour incrementally and end when a complete tour is created. NN, DENN and GREEDY are all classical construction heuristics for the TSP. We refer readers to Section 3.2 in [90] for a detailed description of the three purely greedy algorithms and the results of empirical investigation of the TSP. For the CTSPP, GRASP is an algorithm that combines greediness and randomness. The motivation of using a greedy algorithm is that, for a cumula-

tive problem, edges visited earlier are given higher weights when calculating the objective value; thus, the shorter edges should be visited before the longer edges. However, a purely greedy algorithm may lead to myopic behaviour. Therefore, GRASP incorporates a controlled amount of randomness in the greedy algorithm to decrease the probability of local optimality. We also propose IGRASP, which reduces the memory space of GRASP, and our later computational experiments suggest its superiority to GRASP. IGRASP can be seen as a generalised version of the classical NN heuristic, as it incorporates randomness into NN. RIH has been extensively used for the TSP [89, 75], but here it is first applied to the CTSPP to the best of our knowledge. The three tour construction heuristics are demonstrated below and are used to generate initial solutions for the later experiments.

GRASP

In the conventional sense, GRASP consists of a constructive phase to construct an initial solution and an improvement phase to search the neighbourhood [62]. This method has applications in combinational optimisation problems such as the vehicle routing problem (VRP) with time windows [96]. However, in the research on the CTSPP, the term GRASP has only been used in the constructive phase [134, 137, 113]. For consistency, in this chapter, GRASP here only refers to the tour construction heuristic.

For GRASP, we have a parameter α. Before starting the procedure, we build an $n \times (n-1)$ neighbour matrix D which contains the closest, second-closest, $\ldots$, $(n-1)th$ closest vertices to each vertex, as determined using the distance data. Starting from the depot, the search procedure chooses a random vertex from a restricted candidate list (RCL) containing the α closest vertices to the depot. The chosen vertex is added to the tour and is unavailable for selection afterwards. Then the RCL is updated with the α closest vertices to the last added vertex, excluding the vertices that have already been chosen. The size of the RCL γ is constant until the number of remaining vertices is smaller than α, then γ is reduced. The update of the RCL and the random selection are repeated until all vertices have been chosen. Note that γ is always greater than 0 in the process. The parameter α reflects the balance between randomness and greediness; the greater the value of α, the more random the solution. When $\alpha = 1$, GRASP is a completely greedy search, whereas when $\alpha = n - 1$, GRASP is a completely random search. GRASP has been applied to the CTSPP by several scholars such as Salehipour et al. [134] and Silva et al. [137]. The algorithm is given in Algorithm 3.2.1. The tour τ is regarded as an initial solution.

Algorithm 3.2.1

Algorithm: GRASP(α) with a preprocessed $n \times (n-1)$ neighbour matrix

$S \leftarrow \{2, 3, \ldots, n\}$;
$\tau_1 \leftarrow 1$; $k \leftarrow 1$; $\tau \leftarrow \langle \tau_1 \rangle$;
while $S \neq \emptyset$ ***do***
  $RCL \leftarrow S \cap \{\min(\alpha, n-k)$ *vertices closest to* $\tau_k\}$;
  $\tau_{k+1} \leftarrow$*a random vertex selected from the RCL;* $\tau \leftarrow \langle \tau, \tau_{k+1} \rangle$;
  $S \leftarrow S \setminus \{\tau_{k+1}\}$;
  $k \leftarrow k + 1$;
return τ.

IGRASP

Next, IGRASP is proposed, in which another parameter $\beta (\beta < n-1)$ is used. For GRASP, each row of the neighbour matrix contains the $n-1$ closest vertices, while for IGRASP, only the β closest vertices to each vertex are listed, which means that each row of the neighbour matrix D' only contains β vertices. This reduces the memory space. Similarly to in GRASP, a random vertex is chosen from the RCL, but for IGRASP, the construction of the RCL is slightly different. For the last-added vertex, we find β vertices in the corresponding row in the neighbour matrix and delete the elements that have already been chosen. The RCL consists of the remaining vertices. This means that the RCL can be empty if all β vertices have been chosen. In this case, the first available nearest vertex is chosen. For IGRASP, γ is still defined as the size of the RCL, but γ is dynamic and can be 0. The process is repeated until a complete initial solution τ is generated. When $\beta = 1$, IGRASP is the NN heuristic. The introduction of randomness can remedy the problems of pure greediness to some degree. The algorithm of IGRASP, which is applied to the CTSPP for the first time, is given in Algorithm 3.2.2.

Algorithm 3.2.2

Algorithm: IGRASP(β) with a preprocessed $n \times \beta$ neighbour matrix

$S \leftarrow \{2, 3, \ldots, n\}$;
$\tau_1 \leftarrow 1$; $k \leftarrow 1$; $\tau \leftarrow \langle \tau_1 \rangle$;
while $S \neq \emptyset$ ***do***
  $RCL \leftarrow S \cap \{\beta$ *vertices in the* τ_k*th row in* $D'\}$;
  if $RCL \neq \emptyset$ ***then***
    $\tau_{k+1} \leftarrow$ *a random vertex selected from the* RCL;
  else
    $\tau_{k+1} \leftarrow$ *the first available nearest vertex to* τ_k;
  $\tau \leftarrow \langle \tau, \tau_{k+1} \rangle$; $S \leftarrow S \setminus \{\tau_{k+1}\}$;
  $k \leftarrow k + 1$;
return τ.

We use the distance matrix C as an instance to illustrate the difference between GRASP and IGRASP. The neighbour matrices for GRASP and IGRASP are given as D and D' respectively as follows.

$$C = \begin{pmatrix} 0 & 4 & 2 & 7 & 1 & 3 \\ 4 & 0 & 3 & 2 & 6 & 5 \\ 2 & 3 & 0 & 5 & 8 & 1 \\ 7 & 2 & 5 & 0 & 4 & 3 \\ 1 & 6 & 8 & 4 & 0 & 7 \\ 3 & 5 & 1 & 3 & 7 & 0 \end{pmatrix} \quad D = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \end{array} \begin{array}{c} \begin{array}{ccccc} v_5 & v_3 & v_6 & v_2 & v_4 \end{array} \\ \begin{pmatrix} v_5 & v_3 & v_6 & v_2 & v_4 \\ v_4 & v_3 & v_1 & v_6 & v_5 \\ v_6 & v_1 & v_2 & v_4 & v_5 \\ v_2 & v_6 & v_5 & v_3 & v_1 \\ v_1 & v_4 & v_2 & v_6 & v_3 \\ v_3 & v_1 & v_4 & v_2 & v_5 \end{pmatrix} \end{array} \quad D' = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \end{array} \begin{array}{c} \begin{array}{cc} v_5 & v_3 \end{array} \\ \begin{pmatrix} v_5 & v_3 \\ v_4 & v_3 \\ v_6 & v_1 \\ v_2 & v_6 \\ v_1 & v_4 \\ v_3 & v_1 \end{pmatrix} \end{array} \tag{3.5}$$

Assume $\alpha = 2$ for GRASP and $\beta = 2$ for IGRASP. The tour construction including the added vertex and updated RCL in each step is presented in Table. 3.1. The tours constructed are $(v_1, v_3, v_2, v_6, v_5, v_4)$ and $(v_1, v_3, v_6, v_4, v_2, v_5)$ for GRASP and IGRASP respectively.

RIH

The procedure of this heuristic is straightforward: starting from the depot,

Table 3.1: Comparison of tour construction for GRASP ($\alpha = 2$) and IGRASP ($\beta = 2$)

GRASP			IGRASP		
Added vertex	Updated RCL	γ	Added vertex	Updated RCL	γ
$\tau_1 = v_1$	$\{v_5, v_3\}$	2	$\tau_1 = v_1$	$\{v_5, v_3\}$	2
$\tau_2 = v_3$	$\{v_6, v_2\}$	2	$\tau_2 = v_3$	$\{v_6\}$	1
$\tau_3 = v_2$	$\{v_4, v_6\}$	2	$\tau_3 = v_6$	$\emptyset$	0
$\tau_4 = v_6$	$\{v_4, v_5\}$	2	$\tau_4 = v_4$	$\{v_2\}$	1
$\tau_5 = v_5$	$\{v_4\}$	1	$\tau_5 = v_2$	$\emptyset$	0
$\tau_6 = v_4$			$\tau_6 = v_5$		

it first randomly chooses a vertex as the second node. Then, in each iteration, it randomly selects a vertex that has not been in the partially built solution, and inserts the vertex into the route so that the increment in the cumulative objective value is minimised. The process is repeated until a complete route is constructed.

3.2.3 Classical Simple Tour Improvement Heuristics

Tour improvement heuristics start from an initial tour and modify the current tour repeatedly to find a better solution. A well-known tour improvement heuristic is k-opt [106], where k non-adjacent edges are removed and another k edges are reconnected. The main k-opt heuristics used in combinatorial optimisation are 2-opt [44] and 3-opt [21]. Lin [105] suggested that 4-opt was not an ideal option when considering the trade-off between the tour quality and computational time. To obtain a better trade-off, Bentley [17] designed 2.5-opt heuristic, and Or [119] proposed a restricted version of 3-opt (also called Or-opt). In this chapter, the term 3-opt refers to Or-opt. The swap heuristic is a straightforward tour improvement heuristic that exchanges the positions of each pair of vertices in the tour [79]. Further, we consider another three heuristics, 1-insertion, 2-insertion and 3-insertion, which simply reinsert one, two and three consecutive vertices respectively into the tour.

Here we analyse seven classical simple tour improvement heuristics: 2-opt, 3-opt, swap, swap-adjacent, 1-insertion, 2-insertion and 3-insertion. When updating the objective value using simple heuristics such as 2-opt and 3-opt for the TSP, the calculation can be obtained within a constant time. In contrast, for the CTSPP, the calculation is more complicated because the coefficients of the reconnected edges are changed. In the worst case, the update requires $O(n)$ time for one change. However, when we use the self-generated data memory to calculate the change in

the value of the objective function, Δf, for all simple heuristics, we can obtain Δf in constant time. Hereafter, this method is called the *change value method*. If $\Delta f < 0$, we conclude that a better solution has been found.

Swap

The swap heuristic exchanges the positions of each pair of vertices in the initial tour. Denote $\tau = \langle \tau_1, \tau_2, \ldots, \tau_{i-1}, \tau_i, \tau_{i+1}, \ldots, \tau_{j-1}, \tau_j, \tau_{j+1}, \ldots, \tau_n \rangle$ as the initial solution, then $\tau' = \langle \tau_1, \tau_2, \ldots, \tau_{i-1}, \tau_j, \tau_{i+1}, \ldots, \tau_{j-1}, \tau_i, \tau_{j+1}, \ldots, \tau_n \rangle$ is the new solution after swapping the positions of τ_i and τ_j. Then the change in the value of the objective function is $\Delta f = (c_{\tau_{i-1}\tau_j} - c_{\tau_{i-1}\tau_i}) \times (n-i+1) + (c_{\tau_j\tau_{i+1}} - c_{\tau_i\tau_{i+1}}) \times (n-i) + (c_{\tau_{j-1}\tau_i} - c_{\tau_{j-1}\tau_j}) \times (n-j+1) + (c_{\tau_i\tau_{j+1}} - c_{\tau_j\tau_{j+1}}) \times (n-j)$, where Δf can be obtained in $O(1)$. Thus, exploring the entire swap neighbourhood requires $O(n^2)$ time.

Swap-adjacent (Swap-ad)

The swap-adjacent heuristic exchanges the positions of each pair of adjacent vertices in the initial tour. This heuristic is a subset of the swap heuristic. After one swapping operation, the change in the value of the objective function is $\Delta f = (c_{\tau_{i-1}\tau_{i+1}} - c_{\tau_{i-1}\tau_i}) \times (n-i+1) + (c_{\tau_{i+1}\tau_i} - c_{\tau_i\tau_{i+1}}) \times (n-i) + (c_{\tau_i\tau_{i+2}} - c_{\tau_{i+1}\tau_{i+2}}) \times (n-i-1)$. Similarly, Δf can be obtained in $O(1)$. Thus, the entire neighbourhood is explored in $O(n)$ time.

2-opt

The 2-opt heuristic removes each pair of non-adjacent edges and reconnects the vertices. Denote $\tau = \langle \tau_1, \ldots, \tau_{i-1}, \tau_i, \tau_{i+1}, \tau_{i+2}, \ldots, \tau_{j-1}, \tau_j, \tau_{j+1}, \ldots, \tau_n \rangle$ as the initial solution, then $\tau' = \langle \tau_1, \ldots, \tau_{i-1}, \tau_i, \tau_j, \tau_{j-1}, \ldots, \tau_{i+2}, \tau_{i+1}, \tau_{j+1}, \ldots, \tau_n \rangle$ is the new solution after completing one operation in Figure 3.1, which is taken from the research of Salehipour et al. [134].

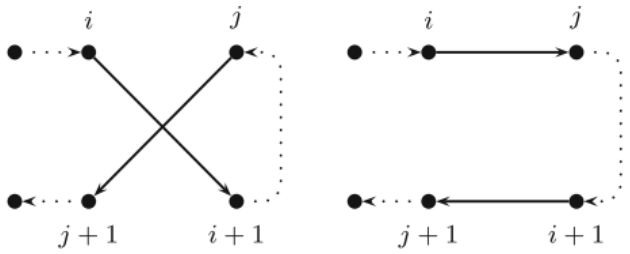


Figure 3.1: Demonstration of 2-opt

Because part of the solution is reversed, the calculation of 2-opt is more complicated than that of the swap and swap-adjacent heuristics. Mladenović et al. [113] explored the complete 2-opt neighbourhood in $O(n^2)$ time by building data

structures and preprocessing. In this chapter, the calculation is simplified using the *change value method*, thus saving preprocessing time. After one operation, the change in the value of the objective function is Δf, which is calculated as follows.

$$\begin{aligned}\Delta f =&(c_{\tau_i\tau_j} - c_{\tau_i\tau_{i+1}}) \times (n-i) + (c_{\tau_{i+1}\tau_{j+1}} - c_{\tau_j\tau_{j+1}}) \times (n-j) \\ &+ \sum_{t=i+1}^{t=j-1} c_{\tau_{t+1}\tau_t} \times (n+t-i-j) - \sum_{t=i+1}^{t=j-1} c_{\tau_t\tau_{t+1}} \times (n-t)\end{aligned} \tag{3.6}$$

Given a fixed i and j, calculating the new Δf with j increased by 1 only requires $O(1)$ time. Thus, for a fixed i, the calculation of Δf for all j requires $O(n)$ time, and the complexity of exploring the entire 2-opt neighbourhood is $O(n^2)$.

3-opt

The 3-opt heuristic removes each triplet of non-adjacent edges and reconnects the vertices with another three edges. There are several different types of connection. In this chapter, only one type of connection is considered to retain the orientation of the initial solution. The operation is shown in Figure 3.2, which is also illustrated in the research of Salehipour et al. [134]. Denote $\tau = \langle \tau_1, \tau_2, \ldots, \tau_i, \tau_{i+1}, \ldots, \tau_j, \tau_{j+1}, \ldots, \tau_k, \tau_{k+1}, \ldots, \tau_n \rangle$ as the initial solution, then $\tau' = \langle \tau_1, \tau_2, \ldots, \tau_i, \tau_{j+1}, \ldots, \tau_k, \tau_{i+1}, \ldots, \tau_j, \tau_{k+1}, \ldots, \tau_n \rangle$ is the new solution after one operation. Essentially, this operation moves $(j-i)$ consecutive vertices $\tau_{i+1}, \ldots, \tau_j$ to the right of τ_k. The change in the value of the objective function is similar to that of 2-opt. Given a fixed i,j and k, calculating the new Δf with k increased by 1 requires a constant time. Thus, for a fixed i and j, the calculation of Δf for all k requires $O(n)$ time, and the complexity of exploring the entire 3-opt neighbourhood is $O(n^3)$.

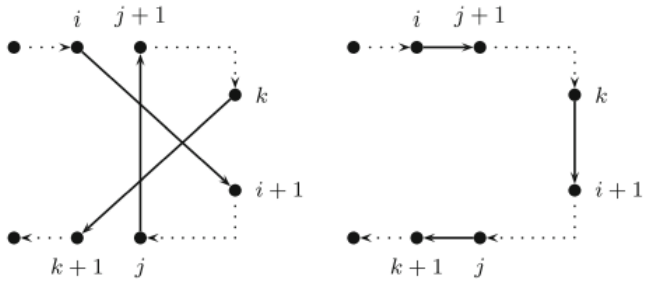


Figure 3.2: Demonstration of 3-opt

1-insertion, 2-insertion, and 3-insertion (1-in, 2-in, and 3-in)

These three heuristics respectively reinsert one, two and three consecutive vertices into another position in the tour, which can be seen as special cases of 3-opt. For example, 3-insertion can be viewed as the case of 3-opt with the index j set to $i + 3$. The complexity of exploring the entire insertion neighbourhood is $O(n^2)$.

3.3 Heuristics Based on Dynamic Programming

We propose three simple heuristics for tour improvement: the pyramidal heuristic, chains heuristic, and sliding window heuristic. These three heuristics are all based on dynamic programming and have simple procedures.

3.3.1 Pyramidal Heuristic

As discussed in Chapter 2, pyramidal tours have important properties in the TSP. The TSP with a Demidenko matrix is pyramidally solvable, which means that an optimal tour for the Demidenko TSP can be found in the set of pyramidal tours [54]. The dynamic programming heuristic in this sub-section is based on the pyramidal neighbourhood search.

The neighbourhood search starts from a candidate solution and then iteratively moves to a neighbour in the neighbourhood. A neighbourhood is the set of all potential solutions based on the neighbourhood relation. The pyramidal neighbourhood contains all neighbours that have pyramidal permutations.

The permutation $\pi = \langle 1, \pi_2, \ldots, \pi_a, \ldots, n, \ldots, \pi_b, \ldots, \pi_n \rangle$ is called a pyramidal permutation if $1 < \pi_2 < \cdots < \pi_a < n$ and $n > \pi_b > \cdots > \pi_n$. In other words, in a pyramidal permutation, part of the indices are placed in increasing order up to n, then the remaining indices are placed in decreasing order. For example, $\pi = \langle 1, 3, 5, 4, 2 \rangle$ is a pyramidal permutation, but $\pi = \langle 1, 3, 5, 2, 4 \rangle$ is not a pyramidal permutation. A pyramidal neighbourhood contains 2^{n-1} neighbours following the pyramidal permutations. Given a tour τ, let us denote $\tau(\pi)$ as its pyramidal neighbour. For example, assume a tour is $\tau = \langle 1, 2, 5, 4, 3 \rangle$ and the pyramidal permutation is $\pi = \langle 1, 3, 5, 4, 2 \rangle$, then the pyramidal neighbour is $\tau(\pi) = \langle 1, 5, 3, 4, 2 \rangle$. Let us denote $N_P(\tau) = \{\tau(\pi) : \pi \in N\}$ as the pyramidal neighbourhood of τ, assuming that N is the set of all pyramidal permutations.

For the TSP, an $O(n^2)$ algorithm based on dynamic programming can be

used to find the optimal pyramidal neighbour. Gilmore et al. [72] applied pyramidal tours to a large neighbourhood search for the TSP. In this chapter, the dynamic-programming-based pyramidal neighbourhood search is applied to the CTSPP for the first time, and the optimal pyramidal neighbour can be found within polynomial time $O(n^3)$ using the dynamic programming recursions.

Lemma 3.3.1 *For the CTSPP, the optimal pyramidal tour can be found within polynomial time $O(n^3)$ using the dynamic programming algorithm.*

Proof. Construct pyramidal permutations that satisfy the following conditions: the sub-permutation starts from index i and ends with index $j(i < j)$ through the set of indices $\{j+1, j+2, \ldots, n\}$. Note that in the constructed complete pyramidal permutation, the position of index i is p, while the position of index j is $n - j + p + 1$. Denote $V(i, j, p)$ as the smallest cumulative value of the sub-tour following the partially constructed pyramidal permutation described above. Given an initial tour τ, the sub-tour goes from τ_i to τ_j through the set of vertices $\{\tau_{j+1}, \tau_{j+2}, \ldots, \tau_n\}$. Additionally, $c_{i,j}$ is denoted as the travelling time between τ_i and τ_j. $V(j, i, p)$ has a similar definition. Since the pyramidal sub-tour is considered, the index $j+1$ can only be placed next to i or next to j. The range of values of p is determined according to the pyramidal property.

For generality, we here consider the case with an asymmetric matrix. As the time spent between the last customer and the depot is not included in the objective value, we can assume that the travelling time from any vertex to the depot is 0. Then the value $V(1, 1, 1)$ is the optimal objective value, which can be found using:

$$V(1,1,1) = \min\left((n-1) \times c_{1,2} + V(2,1,2), c_{2,1} + V(1,2,1)\right), \tag{3.7}$$

where values $V(2, 1, 2)$ and $V(1, 2, 1)$ are calculated using the following dynamic programming recursions:

$$V(i,j,p) = \min\left((n-p) \times c_{i,j+1} + V(j+1, j, p+1), (j-p) \times c_{j+1,j} + V(i, j+1, p)\right),$$

$$\forall j = 2, \ldots, n-1, i = 1, 2, \ldots, j-1, p = \begin{cases} 1, & \text{if } i = 1; \\ 2 \le p \le i, & \text{if } i = 2, \ldots, j-1. \end{cases} \tag{3.8}$$

$$V(j,i,p) = \min\left((n-p)\times c_{j,j+1} + V(j+1,i,p+1), (j-p)\times c_{j+1,i} + V(j,j+1,p)\right),$$

$$\forall j = 2,\ldots,n-1, i = 1,2,\ldots,j-1, p = \begin{cases} j, & \text{if } i = 1; \\ j-i+1 \leq p \leq j-1, & \text{if } i = 2,\ldots,j-1. \end{cases} \tag{3.9}$$

The Boundary conditions (i.e., $j = n$) are given as follows:

$$V(i,n,p) = (n-p)\times c_{i,n}, \forall i = 1,2,\ldots,n-1, p = \begin{cases} 1, & \text{if } i = 1; \\ 2 \leq p \leq i, & \text{if } i = 2,\ldots,n-1. \end{cases} \tag{3.10}$$

$$V(n,i,p) = (n-p)\times c_{n,i}, \forall i = 1,2,\ldots,n-1, p = \begin{cases} n, & \text{if } i = 1; \\ n-i+1 \leq p \leq n-1, & \text{if } i = 2,\ldots,n-1. \end{cases} \tag{3.11}$$

This completes the proof. □

Based on the above recursions, a local search heuristic with the best improvement strategy is adopted. The algorithm is given in Algorithm 3.3.2 and is called the pure pyramidal heuristic.

Algorithm 3.3.2

Algorithm: Pure pyramidal heuristic with initial solution τ

```
τ' ← τ;
improve ← true;
while improve do
    improve ← false;
    τ'' ← argminN_P(τ') using the dynamic programming recursions;
    if τ'' is better than τ' then
        τ' ← τ'';
        improve ← true;
return τ'.
```

Carlier and Villon [30] developed a heuristic based on the pyramidal neighbourhood combined with cyclic shifts for the TSP, which outperformed the well-

known 2-opt in computational experiments. Deineko and Woeginger [53] also stated that this heuristic was competitive with k-opt for the TSP. We also propose an algorithm combining the pyramidal heuristic with the cyclic shifts for the CTSPP. This is the first time that this type of neighbourhood search has been applied to the CTSPP. As in Section 2.1, a cyclic shift is the operation of rearranging the entries in a permutation, either by moving the last element to the first position and moving all other elements to the next positions, or by performing the inverse operation. Given a permutation containing n elements, a cyclic shift can be repeatedly performed to generate $n-1$ permutations. More specifically, given an identity tour $\tau^0 = \langle \tau_1^0, \tau_2^0, \tau_3^0, \ldots, \tau_n^0 \rangle$, the first cyclic shift is $\tau^1 = \langle \tau_1^0, \tau_3^0, \tau_n^0, \ldots, \tau_2^0 \rangle$, the second cyclic shift is $\tau^2 = \langle \tau_1^0, \tau_4^0, \ldots, \tau_n^0, \tau_2^0, \tau_3^0 \rangle$, and so forth. There are $n-2$ cyclic shifts in total for an n-size tour as the depot can only be placed in the first position. For the identity tour τ^0 and each of its cyclic shifts $\tau^1, \tau^2, \ldots, \tau^{n-2}$, denote $\sigma^i (i = 0, 1, \ldots, n-2)$ as the corresponding optimal pyramidal tour, which can be obtained using the Algorithm in Table 3.3.2. σ^i with the smallest objective value is then chosen as the new identity tour for renumbering nodes. This search is repeated until no better solutions can be found. The algorithm is given in Algorithm 3.3.3. In Section 3.4, two versions of the pyramidal heuristic are experimentally investigated.

Algorithm 3.3.3

Algorithm: Pyramidal heuristic with cyclic shifts with initial solution τ

$\tau' \leftarrow \tau$;
improve $\leftarrow$ *true*;
while *improve* ***do***
  improve $\leftarrow$ *false*;
  $k \leftarrow 0$; $\tau^0 \leftarrow \tau'$; *find cyclic shifts* $\tau^1, \tau^2, \ldots, \tau^{n-2}$;
  while $k \leq n-2$ ***do***
    $\sigma^k \leftarrow$ *Use pure pyramidal heuristic with initial solution* τ^k;
    if σ^k *is better than* τ' ***then***
      $\tau' \leftarrow \sigma^k$;
      improve $\leftarrow$ *true*;
    $k \leftarrow k+1$;
return τ'.

3.3.2 Chains Heuristic

The total number of possible permutations covering n nodes is $(n-1)!$ assuming that the first node (depot) is always node 1, which makes enumeration computationally intractable for large-size problems. However, if a tour is partitioned into a set of chains whose length is a small value k and all chains must be visited in order, then the best tour with this structure can be obtained with much less computation. Here, the length refers to the number of nodes in the chain.

In our algorithm, if n is a multiple of k, then the lengths of all chains are set as k. Otherwise, the tour is partitioned into chains according to the following rules: the length of the first chain and the last chain can be $3, 4, \ldots, k-1, k$, while the length of the middle chains can only be k. The parameter k is set as 6 because this value gave the best performance in preliminary experiments.

We partition the identity permutation $\pi = \langle 1, 2, \ldots, n \rangle$ into m chains as $\pi = \langle CN^1, CN^2, \ldots, CN^i, \ldots, CN^m \rangle$. For chain CN^i let us denote l^i as its length and a^i and b^i as the first index and last index in the chain respectively. Note that $l^i = b^i - a^i + 1$.

The procedure starts from the last chain CN^m. There are $(l^m)!$ permutations of the chain in total. We calculate the cumulative cost for each permutation of the chain and record the best permutation (with the smallest cost) starting from each index t as $P_t^m (a^m \leq t \leq b^m)$. We take the backtrace and consider the penultimate chain CN^{m-1}. There are $(l^{m-1})!$ permutations for this chain, and each permutation can be linked to P_t^m; thus, there are $(l^{m-1})! \times l^m$ combinations in total. Then the cumulative cost is calculated for all these combinations, and the best permutation $P_t^{m-1} (a^{m-1} \leq t \leq b^{m-1})$ is recorded. We repeat the process until we reach the first chain CN^1. Note that when we calculate the costs for the first chain, the cost linked from the depot to the first chain should be included. Using this procedure, we can find the best neighbour in the chains neighbourhood whose size is $(l^1)! \times (l^2)! \times (l^i)! \times (l^m)!$, and the computational complexity of this subroutine is $O(nk!)$.

An example is used to demonstrate the procedure as shown in Figure 3.3. Assume $n = 20, k = 6$, then the chain lengths are 3, 6, 6 and 4 in order, and this neighbourhood contains $(3!) \times (6!) \times (6!) \times (4!)$ neighbours. The procedure starts from the last chain CN^4. There are 4! permutations of the chain in total. We calculate the cumulative cost for each permutation of the chain, and record the best permutation starting from each index t as $P_t^4 (17 \leq t \leq 20)$. For ex-

ample, $P_{18}^4 = \langle 18, 19, 17, 20\rangle$ and $P_{20}^4 = \langle 20, 19, 17, 18\rangle$. We take the backtrace and consider the penultimate chain CN^3. There are 6! permutations for this chain, and each permutation can be linked to $P_t^4(17 \leq t \leq 20)$; thus, there are $(6! \times 4)$ combinations in total. We then calculate the cumulative cost for all these combinations, and record the best permutation $P_t^3(11 \leq t \leq 16)$. For example, to calculate P_{11}^3, we need to calculate the cost of all permutations starting from index 11 linked to the optimal 'tail' recorded, i.e., the costs of permutations $\langle 11, 12, 13, 14, 15, 16\rangle, \ldots \langle 11, 16, 15, 14, 13, 12\rangle$ linked to $P_{17}^4, P_{18}^4, P_{19}^4, P_{20}^4$, and choose the one with the smallest cost as P_{11}^3. The process is repeated until the first chain CN^1 is explored.

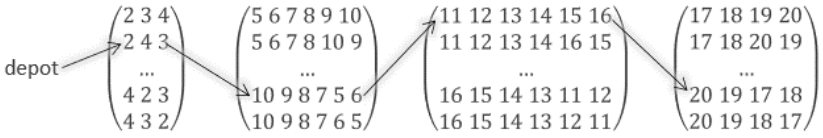


Figure 3.3: Demonstration of chains neighbourhood

We use the subroutine in Algorithm 3.3.4 to obtain an optimal neighbour in the chains neighbourhood. Then we apply the subroutine to a local search heuristic, where the optimal solution of the subroutine is used as a new initial solution and the search is repeated until no better solution is found.

Algorithm 3.3.4

Algorithm: Subroutine of chains neighbourhood search with initial solution τ

Construct list of chains: $CN^1, CN^2, \ldots, CN^m$; $k \leftarrow m - 1$;
Calculate cumulative cost of CN^m *with all permutations and record* $P_t^m(a^m \leq t \leq b^m)$;
while $k \geq 2$ ***do***
  for $(a^k \leq t \leq b^k)$
    Calculate cost of all permutations of CN^k *starting from* t*, linked to* $P_{t'}^{k+1}(a^{k+1} \leq t' \leq b^{k+1})$ *and record* P_t^k *with the smallest cost;*
  $k \leftarrow k - 1$;
for *(all permutations of* CN^1*)*
  Update cost of CN^1 *linked to the depot and the optimal 'tail' recorded above;*
  Determine the minimum cost as the optimal value;
Reconstruct the optimal tour τ' *with the recorded link;*
return τ'.

3.3.3 Sliding Window Heuristic

The pure dynamic programming approach has an exponential time complexity of $O(2^n n^2)$ and therefore cannot be applied to large-size problems. An aggregation strategy with intervals is adopted to reduce the problem size. First, the initial solution is disassembled into a sequence of intervals, each of which is represented as a new customer. Second, we adopt the dynamic programming recursions in Wu's research [147] to find an exact optimal solution for the small-size CTSPP with new sets of customers. The solution obtained can be transformed into a solution for the initial problem by replacing the aggregated customers with the original sub-paths. The solution obtained will not be optimal; instead, it can only be viewed as a heuristic solution for the initial problem. The solution can then be disassembled again to obtain a new small-size problem, and the process is repeated until a certain stopping criterion is satisfied.

To disassemble the tours, a straightforward approach called sliding window aggregation is used. This method was applied to 2-VRP in the research of Deineko et al. [49]. We incorporate the cumulative objective into sliding window aggregation and propose the sliding window heuristic for the CTSPP. We first demonstrate the method of disassembling a tour into intervals. Secondly, we calculate the cumulative value for the connected intervals.

Assume that the initial tour is $x = \langle x_1, x_2, \ldots, x_n \rangle$ with x_1 as the depot. Choose two subsets of vertices S_1 and S_2, each of which contains s consecutive vertices in x. S_1 and S_2 are called the first and second sliding windows in the algorithm respectively. In the first phase, we set $S_1 = \{x_2, x_3, \ldots, x_{s+1}\}$ and $S_2 = \{x_{s+3}, x_{s+4}, \ldots, x_{2s+2}\}$. We delete S_1 and S_2 from the initial tour and add them to the set of vertices in the new CTSPP. The remaining sub-paths in x are viewed as intervals of customers and added to the new CTSPP. In this stage, two intervals are $\langle x_{s+2} \rangle$ and $\langle x_{2s+3}, x_{2s+4}, \ldots, x_n \rangle$. Each interval is considered as an aggregated customer. The new small-size problem includes one depot x_1, $2s$ single customers from S_1 and S_2 and two intervals/aggregated customers. Therefore, the size of the new small problem is $2s + 3$.

Figure 3.4, taken from the work of Deineko et al. [49], demonstrates the first phase of the sliding window aggregation method. To simplify the drawing, we set $s = 3$ and $n = 12$; thus, $S_1 = \{x_2, x_3, x_4\}$ and $S_2 = \{x_6, x_7, x_8\}$. The new small problem has a size of 9 and includes one depot x_1, six single customers: $x_2, x_3, x_4, x_6, x_7, x_8$ and two intervals $\langle x_5 \rangle$, $\langle x_9, x_{10}, x_{11}, x_{12} \rangle$.

Then the dynamic programming method is applied to solve the small-size problem to optimality. Note that the order of vertices within the intervals cannot be changed. However, to cover more neighbours in the search, we consider two directions of each interval. In the example in Figure 3.4, the sub-path in the solutions for the initial problem can be $\langle x_9, x_{10}, x_{11}, x_{12}\rangle$ or $\langle x_{12}, x_{11}, x_{10}, x_9\rangle$, but it cannot be $\langle x_9, x_{11}, x_{10}, x_{12}\rangle$. If the solution to the initial problem is improved, the process of disassembling is restarted from the new solution.

If there is no improvement in the solution, we enter the second phase: re-defining the subsets. In this stage, S_2 is redefined by deleting the m first elements (m is a parameter called step) and adding m new consecutive elements. Figure 3.5, which is also from the research of Deineko et al. [49], illustrates the outcome of sliding S_2 with $m = 1$ during the second phase with subsets $S_1 = \{x_2, x_3, x_4\}$ and $S_2 = \{x_7, x_8, x_9\}$.

The sliding process is repeated until the end of the tour is reached. When all S_2 are enumerated, we redefine S_1 by moving m steps and redefine S_2 to follow S_1 similarly to the process described above. In the process, if a better solution is found, the disassembling process is restarted from the improved solution. After enumerating all possible S_1 and S_2, if no improved solution can be found, the procedure is stopped. The main sliding subroutine is denoted as $S(s, m)$ with two parameters s and m.

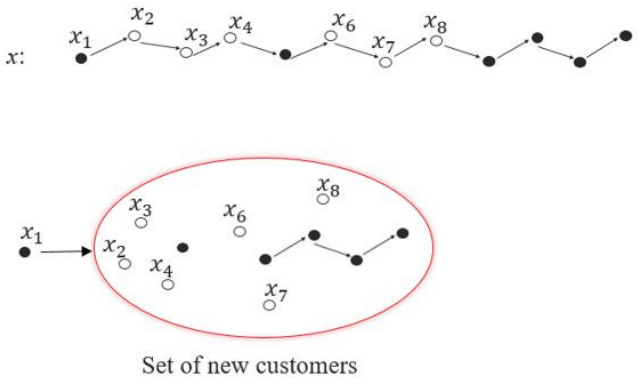


Figure 3.4: Demonstration of the first phase of the sliding window aggregation with $S_1 = \{x_2, x_3, x_4\}$, and $S_2 = \{x_6, x_7, x_8\}$

Figure 3.5: Outcome of redefining S_2 with $m = 1$ in the second phase

In this process, we need to calculate the cumulative value if two intervals are connected. Denote $I = \langle I_1, I_2, \ldots, I_k \rangle$ and $I' = \langle I'_1, I'_2, \ldots, I'_m \rangle$ as the two intervals and $I + I' = \langle I_1, I_2, \ldots, I_k, I'_1, I'_2, \ldots, I'_m \rangle$ as the connected interval. We define three attributes for one interval: $T(I)$, $C(I)$ and $N(I)$. Let $T(I)$ be the travelling time of the interval, $C(I)$ be the cumulative cost of the interval and $N(I)$ be the number of customers of the interval. Again, c_{ij} denotes the travelling time between vertex i and vertex j. Then for the connected intervals, the following three attributes are calculated.

$$\begin{aligned} T(I + I') &= T(I) + T(I') + c_{I_k I'_1} \\ N(I + I') &= N(I) + N(I') \\ C(I + I') &= C(I) + C(I') + (T(I) + c_{I_k I'_1}) \times N(I') \end{aligned} \tag{3.12}$$

These formulas can be applied to both intervals with a single customer and intervals with multiple customers. Silva et al. [137] also adopted similar formulas to evaluate simple moves.

Note that there are two differences between the chains heuristic and the sliding window heuristic: the strategy of reducing the problem size and way of searching. First, for the chains heuristic, each chain is regarded as a small solvable problem, while for the sliding window heuristic, the whole problem is simplified to a small solvable problem. Second, the neighbourhood search is 'internal' for the chains heuristic. The search is conducted within each chain, and the positions of the chains are fixed. In contrast, the sliding window heuristic has an 'external' search, where the positions of the intervals can be changed, but the relative positions of the nodes within each interval are fixed.

3.4 Computational experiments

To test the power of the classical simple heuristics and the proposed heuristics, they are coded in C++ and executed on the Intel Core i5-10600 3.30 GHz processor with 32.0 GB RAM.

3.4.1 Experiment design

The algorithms are tested on both self-generated instances and benchmark instances. As the self-generated instances, we generated 100 geometric instances, i.e., 50 random uniform Euclidean instances and 50 random clustered Euclidean

instances, for each of the problem sizes $n = 50$, 100 and 200. Thus, we generated six instance sets in total. These instances are used in the first experiment. The generation process is as follows.

Random uniform Euclidean instances The city points have two integer coordinates chosen randomly from the uniform distribution $(0, 100]$. Distances are Euclidean distances rounded to the nearest integer.

Random clustered Euclidean instances We set 3, 5 and 10 cluster centres for the problem sizes $n = 50$, 100 and 200 respectively. The coordinates of the cluster centres are chosen randomly from the uniform distribution $(0, 100]$. We first assign a random cluster centre to each city node. Then for each city node, two coordinates are generated from the standard normal distribution and are multiplied by $\frac{0.5\times100}{\sqrt{n}}$, and then added to the coordinates of its centre. Distances are also Euclidean distances rounded to the nearest integer. Johnson and McGeoch [90] used this generation method to compare heuristics for symmetric and asymmetric cases of the TSP.

The benchmark instances we used in the experiments include four sets of databases, each containing 20 symmetric instances, and were generated by Salehipour et al. [134]. The sizes of the four sets are 50, 100, 200 and 500. The benchmark instances are uniform Euclidean instances whose coordinates are chosen randomly from the uniform distribution (0, 100]. Distances are also Euclidean distances rounded to the nearest integer. Detailed information of the test instances is given in Appendix B. Silva et al. [137] also completed computational experiments on these benchmark instances. These instances are used in the second experiment.

Edge-unweighted trees and edge-weighted trees are generated and are tested in the third and fourth experiments respectively. We first use the Prufer code [124] to generate unweighted trees with sizes $n = 50$, 100, 200 and 500. Each instance set has 50 instances. The Prufer code is a unique sequence associated with a tree and can be generated by a simple iterative algorithm. In the generation process, the Prufer code is first initialised as empty. Start with a leaf with the lowest label x, and find the vertex connecting it to the rest of the tree y. Remove x from the tree and add y to the Prufer code. Repeat the process until two nodes are left. An $(n-1) \times (n-1)$ adjacent matrix is then obtained that indicates whether a pair of nodes i and j are connected by an edge. For each unweighted-tree instance, if i and j are connected by an edge, the weight of the edge is 1, otherwise it is 0. Starting from the unweighted-tree instances, we generate weighted-tree instances by assigning a random weight from the range $[1, 20]$ to the non-zero edges. Then

we calculate the distance between any two nodes.

First Experiment

The first experiment is conducted on self-generated general instances. The aim is to test the power of the classical simple tour improvement heuristics and determine the best one to use for comparison with the proposed heuristics in the subsequent experiments. We combine each of the seven simple tour improvement heuristics (swap, swap-ad, 2-opt, 3-opt, 1-in, 2-in and 3-in) with each of the three tour construction heuristics (GRASP, IGRASP and RIH), giving 21 combinations in total. We run the 21 algorithms for each individual instance and adopt the multi-start, i.e., each algorithm is run multiple times and the best result is recorded as the objective value v for each instance. The number of starts is set as the size of the instances.

The objective value v and the running time t measured in seconds are recorded for each instance using each algorithm. Then the best solution among all algorithms is chosen as the comparison base v_{best} for each instance, and the percentage gap $g\% = \frac{v - v_{best}}{v_{best}} \times 100\%$ is calculated as the solution quality. We compare both the solution quality and the running time and investigate whether one algorithm is dominated by another. For each instance set, the average values of $g\%$ and t, respectively denoted as $G\%$ and T, are calculated.

Second Experiment

The second experiment is conducted on benchmark instances. The proposed algorithms are compared with the classical algorithm GRASP-2-opt. Comparing IGRASP and RIH, the findings in the first experiment in Section 3.4.2 suggest that RIH is more efficient when combined with the proposed tour improvement heuristics. Therefore, we conduct experiments on six algorithms: RIH+pure pyramidal heuristic (RIH-PyramidalP), RIH+pyramidal heuristic with cyclic shifts (RIH-PyramidalCS), RIH+chains heuristic (RIH-Chains), RIH+sliding window heuristic with two sets of parameters, $S(2,1)$ and $S(3,1)$, and GRASP-2-opt.

Firstly, we perform tests on the single-start and conduct the domination analysis. Because of the long running time, RIH-PyramidalCS, RIH-$S(2,1)$ and RIH-$S(3,1)$ are not tested on size-500 instances. Secondly, we adopt the multi-start for GRASP-2-opt to ensure the same running time as for the proposed algorithms with the single-start. More specifically, for each instance, GRASP-2-opt runs multiple times and stops when the running time reaches the time used by the single-start proposed algorithms, then the best solution obtained is recorded as v.

Silva et al. [137] obtained the best known solution bk for benchmark in-

stances. We thus calculate the percentage excess over the best known solution, $bk\% = \frac{v-bk}{bk} \times 100\%$, as the measurement of solution quality for each instance. Similarly, the average values of $bk\%$ for each instance set, denoted as $BK\%$, is calculated. The best known solutions are shown in Appendix B. Note that the best known solutions for the 50-city instances are the optimal solutions.

In addition, we conduct further analysis on the RIH+sliding window heuristic. More specifically, we investigate the performance with increasing number of starts. We also investigate the solution quality and running time for RIH+sliding window heuristic for the settings $S(2,1)$ and $S(3,1)$.

Third Experiment

The third experiment is conducted on self-generated edge-unweighted trees. As is known, the CTSPP on edge-unweighted trees is solvable. We conduct the experiments on this special instance set to demonstrate the high quality of the proposed algorithms, which are again compared with GRASP-2-opt. Firstly, we perform tests on the single-start. Then we adopt the multi-start for GRASP-2-opt to ensure the same running time as for the single-start proposed algorithms. In both cases, we calculate the percentage gap over the optimal solution, $g\% = \frac{v-v_{opt}}{v_{opt}} \times 100\%$, for each single instance, and then calculate $G\%$, the average values of $g\%$ for each instance set.

Fourth Experiment

The fourth experiment is conducted on edge-weighted trees. The operations are similar to those in the second experiment. Experiments are conducted on five algorithms: RIH-PyramidalP, RIH-Chains, RIH-$S(2,1)$, RIH-$S(3,1)$ and GRASP-2-opt. Firstly, we perform tests on the single-start and choose the best solution among all algorithms as the comparison base v_{best} for each instance, and we calculate the percentage gap, $g\% = \frac{v-v_{best}}{v_{best}} \times 100\%$, as the solution quality. We compare both solution quality and running time and test whether one algorithm is dominated by another. Then the multi-start is adopted for GRASP-2-opt to ensure the same running time as for the proposed algorithms, and v_{best} is again used as the comparison base to calculate the percentage gap $g\%$. Similarly, the average values of $g\%$ for each instance set, denoted as $G\%$, is calculated.

3.4.2 Experiment Results

First Experiment

We present the experimental results of the classical simple heuristics on

self-generated uniform instances and clustered instances in Tables 3.2 and 3.3 respectively, which show the average solution quality $G\%$ and running time T for each instance set.

One heuristic is said to be dominated by another if the other heuristic generates the same solutions with less time, or generates better solutions with the same time, or is superior in both solution quality and running time. Taking as an example the heuristics combined with IGRASP on 200-city instances, the 1-in, 2-in, 3-in and swap heuristics are all dominated by the swap-ad heuristic. For each instance set, the algorithms that cannot be dominated are underlined. Figure 3.6 shows the trade-off between the average solution quality and running time for the 21 algorithms on 50-city random uniform instances as well as the domination relationships. The figure illustrates that four algorithms (IGRASP-swap-ad, RIH-swap-ad, RIH-1-in, RIH-2-opt) cannot be dominated for this instance set.

The experimental results suggest that the simple heuristics have different powers. First, we focus on the tour construction heuristics. When combined with GRASP, nearly all algorithms (except GRASP-swap-ad) are dominated by other algorithms. This suggests that IGRASP and RIH are superior to the widely used GRASP for the CTSPP. Comparing IGRAS and RIH, we find that for IGRASP only swap-ad and 2-opt are ideal options, while when using RIH for tour construction, fewer tour improvement heuristics will be dominated than for IGRASP. This indicates that RIH is a better heuristic than IGRASP for generating initial solutions. We next focus on the effectiveness of the tour improvement heuristics. The results suggest that 3-opt is dominated by the other tour improvement heuristics regardless of the tour construction heuristic it is combined with. Among the remaining classical simple tour improvement heuristics, although there is no evident domination relationship, 2-opt obtains much better solutions for each instance set with slightly increased running time. Therefore, 2-opt can be seen as the 'best' heuristic among the classical simple heuristics, and the proposed algorithms are thus compared with GRASP-2-opt in the second experiment.

The findings also give guidance for the better selection and combination of simple heuristics for the future study of metaheuristics for the CTSPP. For example, removing 3-opt from the list of neighbourhood structures and replacing GRASP with IGRASP or RIH may improve the efficiency of metaheuristics. In addition, the experimental results can help identify which algorithm works best for problems with different instance characteristics. For example, for a 100-size

Table 3.2: Experimental results of seven classical simple heuristics combined with three tour construction heuristics on self-generated random uniform instances: average solution quality and running time for each instance set

		$n = 50$		$n = 100$		$n = 200$	
		T	$G\%$	T	$G\%$	T	$G\%$
GRASP	2-opt	0.0058	3.48	0.081	3.82	1.16	5.04
	3-opt	0.0525	12.73	11.73	11.04	58.51	11.47
	swap-ad	0.0005	12.10	0.002	14.07	0.01	17.01
	swap	0.0045	12.24	0.062	13.95	0.91	16.02
	1-in	0.0023	16.21	0.036	16.81	0.59	18.08
	2-in	0.0023	18.09	0.028	19.15	0.46	20.65
	3-in	0.0015	20.61	0.020	21.36	0.31	23.24
IGRASP	2-opt	0.0035	1.39	0.049	0.51	0.67	0.15
	3-opt	0.0271	4.91	0.811	3.71	26.12	3.45
	swap-ad	0.0005	4.33	0.002	4.94	0.01	5.77
	swap	0.0020	5.63	0.022	6.10	0.29	7.01
	1-in	0.0016	5.77	0.020	6.02	0.28	6.80
	2-in	0.0014	6.29	0.015	6.42	0.24	7.20
	3-in	0.0013	6.58	0.012	6.82	0.16	7.79
RIH	2-opt	0.0029	0.43	0.032	1.02	0.38	2.32
	3-opt	0.0282	1.80	0.763	1.40	25.44	1.77
	swap-ad	0.0006	2.56	0.003	3.86	0.02	5.03
	swap	0.0013	2.77	0.013	3.91	0.15	4.82
	1-in	0.0012	2.31	0.012	3.79	0.13	4.70
	2-in	0.0018	2.58	0.016	3.15	0.19	3.90
	3-in	0.0013	2.81	0.013	3.78	0.13	4.43

Table 3.3: Experimental results of seven classical simple heuristics combined with three tour construction heuristics on self-generated random clustered instances: average solution quality and running time for each instance set

		$n = 50$		$n = 100$		$n = 200$	
		T	$G\%$	T	$G\%$	T	$G\%$
GRASP	2-opt	0.0092	3.33	0.1201	3.16	1.83	3.73
	3-opt	0.0936	9.64	2.9993	8.77	105.31	7.97
	swap-ad	0.0005	24.35	0.0024	30.39	0.01	36.19
	swap	0.0078	12.20	0.1145	13.68	1.79	15.33
	1-in	0.0046	17.31	0.0743	19.37	1.14	22.80
	2-in	0.0045	19.04	0.0626	20.58	1.03	22.07
	3-in	0.0037	22.68	0.0522	24.36	0.87	25.13
IGRASP	2-opt	0.0059	2.31	0.0739	1.82	1.15	2.08
	3-opt	0.0471	7.06	1.4977	6.32	57.71	5.56
	swap-ad	0.0005	10.16	0.0022	13.22	0.01	17.17
	swap	0.0039	9.24	0.0517	10.43	0.81	12.81
	1-in	0.0030	9.25	0.0397	9.92	0.59	11.94
	2-in	0.0025	9.85	0.0343	10.70	0.54	11.71
	3-in	0.0020	11.74	0.0245	12.27	0.42	13.43
RIH	2-opt	0.0024	0.06	0.0319	0.05	0.47	0.06
	3-opt	0.0229	0.99	0.6360	0.97	23.25	0.96
	swap-ad	0.0007	1.14	0.0037	1.63	0.02	2.16
	swap	0.0014	1.11	0.0155	1.58	0.20	1.76
	1-in	0.0011	1.15	0.0097	1.48	0.09	2.01
	2-in	0.0014	1.29	0.0124	1.54	0.15	1.83
	3-in	0.0011	1.43	0.0104	1.72	0.11	2.07

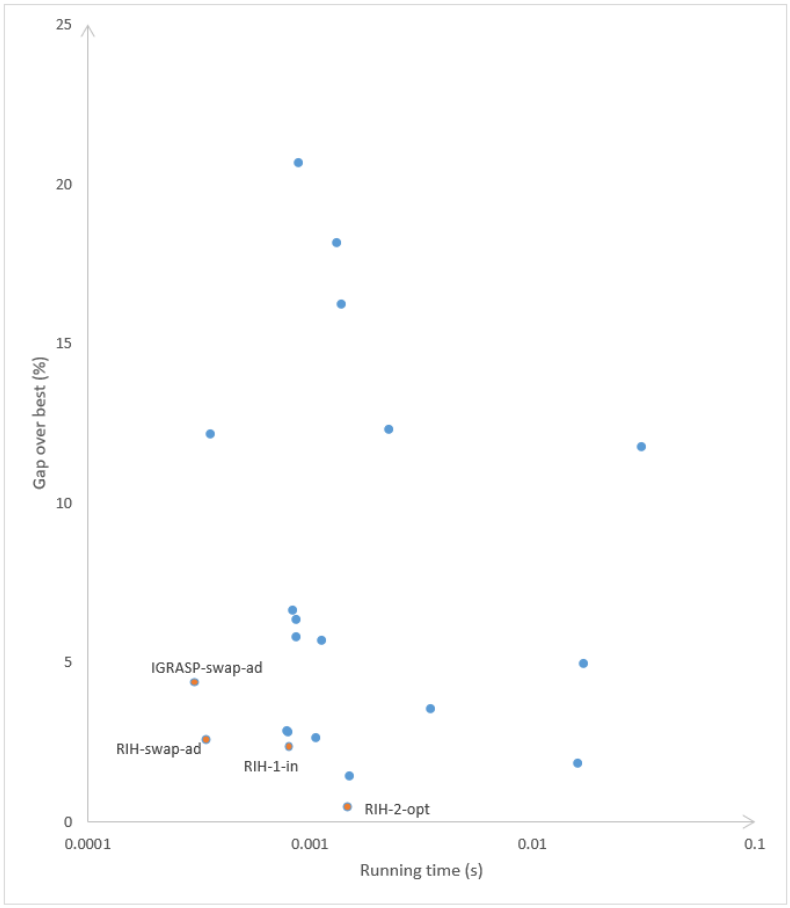


Figure 3.6: Experimental results on 50-size self-generated uniform instances: trade-off between solution quality and running time for 21 algorithms

clustered instance, the best option will be to use RIH for tour construction and 2-opt, swap-ad and 1-in for tour improvement. However, for a 200-size uniform instance, the best option will be to use RIH for tour construction and 2-opt, swap-ad, 1-in, 2-in and 3-in for tour improvement.

Second Experiment

We present the experimental results with the single-start on benchmark instances in Table 3.4 and Figure 3.7. The results suggest that RIH-PyramidalCS is dominated by the other proposed algorithms for instances of all sizes. This result indicates that the pyramidal neighbourhood combined with cyclic shifts does not perform as well for the CTSPP as it does for the TSP, but the pure pyramidal heuristic is promising for the CTSPP. The remaining proposed algorithms obtain better solutions than GRASP-2-opt but with a longer running time.

To remove the trade-off effect and make further comparisons, the multi-start is adopted for GRASP-2-opt to ensure the same running time as for the proposed algorithms with the single-start. For instance, when the size is 50, the average running time of RIH-Pyramidal is 0.0006 s; thus, GRASP-2-opt runs multiple times until the total time used is 0.0006 s, and the best solution obtained is recorded. Because RIH-PyramidalCS has already been shown to be dominated, it is removed from the subsequent experiments.

The results with controlled running time are given in Table 3.5. This table shows the gap above the best known solutions when using the proposed algorithms and GRASP-2-opt and records the number of instances (out of 20) for which the proposed algorithms obtain better solutions than GRASP-2-opt. It is shown that the proposed algorithms perform better as the instance size increases. RIH-$S(2,1)$ outperforms GRASP-2-opt for all experiment sizes. RIH-Chains starts to outperform GRASP-2-opt from size $n = 200$ and its performance continues to improve as the size increases. When the size is greater than 50, RIH-PyramidalP outperforms GRASP-2opt, which is similar to the performance of RIH-$S(3,1)$. The results suggest that the greater the instance size, the better the performance of the proposed algorithms over GRASP-2-opt.

Table 3.4: Experimental results of proposed algorithms and GRASP-2-opt with single-start on benchmark instances: average solution quality and running time for each instance set

	$n = 50$		$n = 100$		$n = 200$		$n = 500$	
	T	$BK\%$	T	$BK\%$	T	$BK\%$	T	$BK\%$
RIH-PyramidalCS	0.0951	4.74	2.615	7.15	75.055	10.83		
RIH-PyramidalP	0.0006	11.98	0.005	14.44	0.046	16.19	0.77	16.86
RIH-Chains	0.0543	8.98	0.126	12.95	0.373	13.82	1.84	14.76
RIH-$S(3,1)$	0.3253	2.70	3.256	3.38	36.109	5.41		
RIH-$S(2,1)$	0.0624	3.09	0.649	4.61	6.437	6.47		
GRASP-2-opt	0.0002	22.64	0.001	23.40	0.008	24.06	0.12	28.27

Additionally, we analyse how the sliding window heuristic performs as the number of starts increases. The chosen numbers of starts are 1, 5, 10, 15, 20, 25, and 30. Figure 3.8 presents the results of RIH-$S(2,1)$ on 50-city benchmark instances. We record the percentage excess over the optimum for each instance and calculate the average excess. The figure shows that the average excess is higher than 3% with

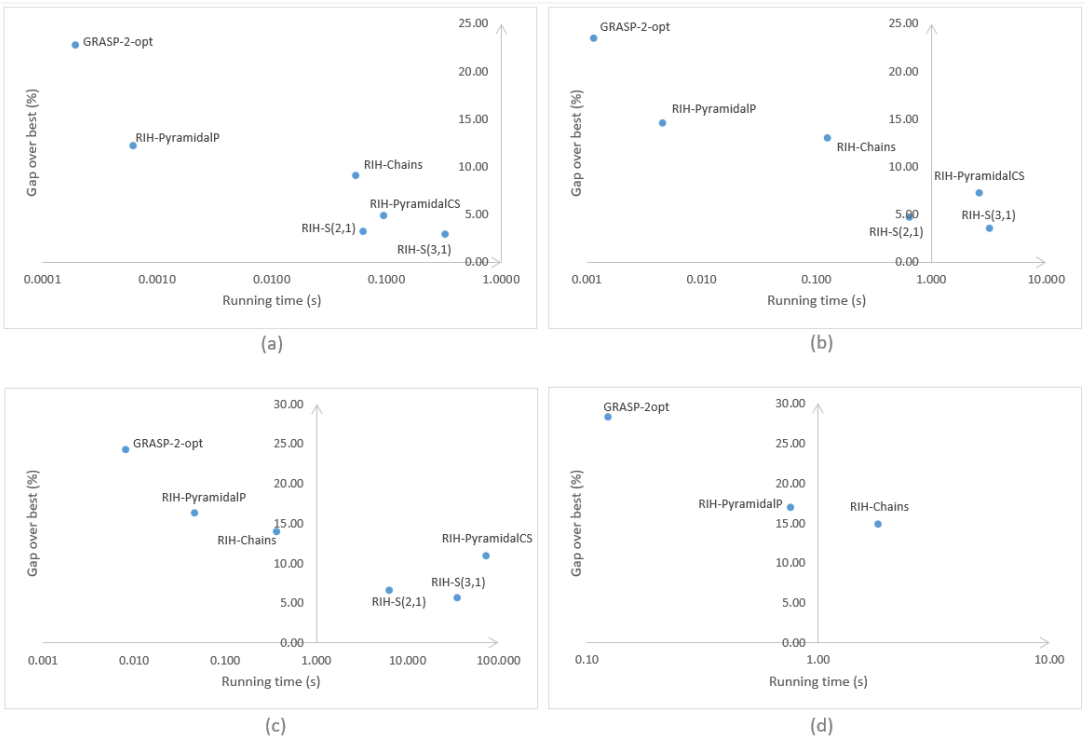


Figure 3.7: Comparison of results of proposed algorithms and GRASP-2-opt on benchmark instances with single-start: (a) 50-size instances; (b) 100-size instances; (c) 200-size instances; (d) 500-size instances

Table 3.5: Comparison of results of proposed algorithms and GRASP-2-opt on benchmark instances with controlled running time

		$BK\%$ of proposed	$BK\%$ of GRASP-2-opt	#better
$n = 50$	RIH-PyramidalP	11.98	11.28	9
	RIH-Chains	8.98	3.46	3
	RIH-$S(3,1)$	2.70	1.91	9
	RIH-$S(2,1)$	3.09	3.35	13
$n = 100$	RIH-PyramidalP	14.44	15.74	12
	RIH-Chains	12.95	9.33	4
	RIH-$S(3,1)$	3.38	6.17	17
	RIH-$S(2,1)$	4.61	8.11	18
$n = 200$	RIH-PyramidalP	16.19	19.20	14
	RIH-Chains	13.82	15.78	16
	RIH-$S(3,1)$	5.41	11.50	20
	RIH-$S(2,1)$	6.47	12.82	20
$n = 500$	RIH-PyramidalP	16.86	23.09	18
	RIH-Chains	14.76	21.47	18

only one start but is almost 0 with 20 starts. The numbers of instances solved to optimality are found to be 3, 10, 11, 15, 16, 18 and 18 for 1, 5, 10, 15, 20, 25 and 30 starts, respectively, as also shown in the figure. This suggests that the multi-start method is very important for generating good solutions. Figure 3.9 compares the gap and the running time for the different settings. As expected, the improvement in the gap obtained by RIH-$S(3,1)$ is at the cost of increased computational time. Figure 3.10 shows the average excess over the optimum plotted against the average running time and demonstrates that there is no domination relationship between different settings.

Third Experiment

The experimental results with the single-start on unweighted trees are presented in Table 3.6. As $G\%$ here is the gap over the optimum, the low values of $G\%$ suggest that the proposed algorithms, especially RIH-$S(2,1)$ and RIH-$S(3,1)$, produce high-quality solutions on this special solvable case. Similar to in the second experiment, there is no domination relationship among the algorithms. The proposed algorithms obtain better solutions but with a longer running time. We also adopt the multi-start for GRASP-2-opt to control the running time. The re-

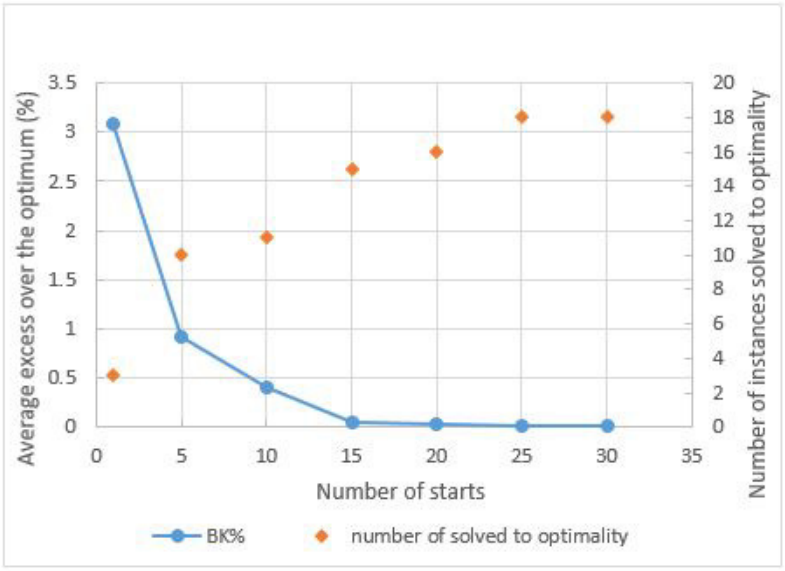


Figure 3.8: Performance of RIH-$S(2,1)$ with increasing number of starts on 50-size benchmark instances

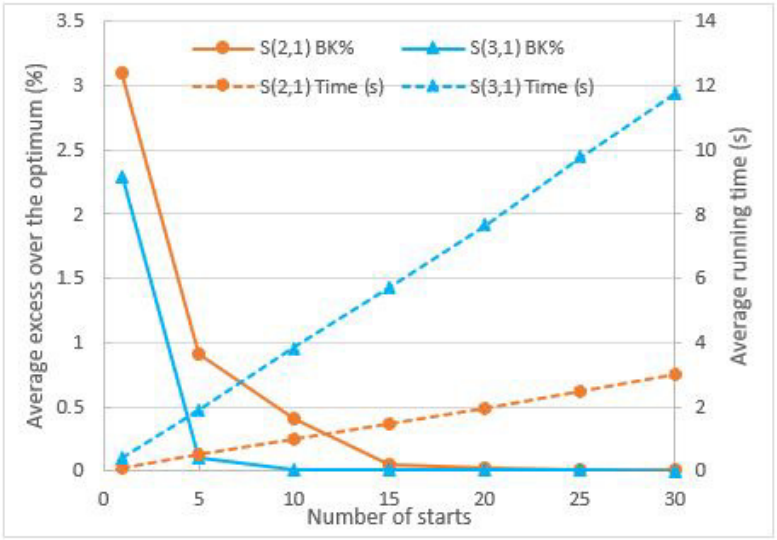


Figure 3.9: Comparison of performance of RIH-$S(2,1)$ and RIH-$S(3,1)$ with increasing number of starts on 50-size benchmark instances

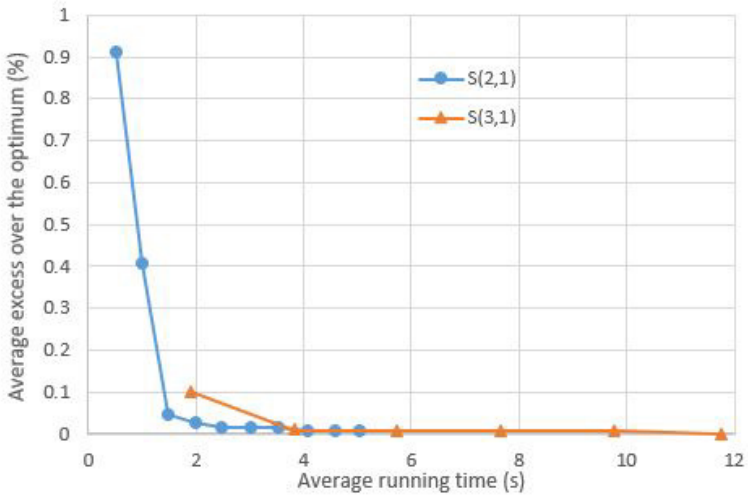


Figure 3.10: Comparison of solution gap plotted against running time for RIH-$S(2,1)$ and RIH-$S(3,1)$ on 50-size benchmark instances

sults on unweighted trees are given in Table 3.7. This table demonstrates that all the proposed algorithms perform better than GRASP-2-opt when the time is controlled, except for RIH-Chains for $n = 50$. The number of instances (out of 50) in which the proposed algorithms obtain better solutions than GRASP-2-opt also suggests that the proposed algorithms have better performance than the classical GRASP-2-opt on the special case of unweighted trees.

Table 3.6: Experimental results of the proposed algorithms and GRASP-2-opt with single-start on unweighted-tree instances: average solution quality and running time for each instance set

	$n = 50$		$n = 100$		$n = 200$		$n = 500$	
	$T(s)$	$G\%$	$T(s)$	$G\%$	$T(s)$	$G\%$	$T(s)$	$G\%$
RIH-PyramidalP	0.0005	2.30	0.003	2.56	0.030	3.13	0.517	3.18
RIH-Chains	0.0347	2.24	0.088	2.54	0.211	3.09	0.999	3.17
RIH-$S(3,1)$	0.1429	0	1.345	0	13.671	0.02		
RIH-$S(2,1)$	0.0253	0	0.230	0.03	2.446	0.03		
GRASP-2-opt	0.0002	13.66	0.001	11.41	0.007	9.95	0.129	8.50

Fourth Experiment

The experimental results on weighted-trees instances with the single-start and multi-start are presented in Tables 3.8 and 3.9 respectively. The findings

Table 3.7: Comparison of results of proposed algorithms and GRASP-2-opt on unweighted-tree instances with controlled running time

		$G\%$ of proposed	$G\%$ of GRASP-2-opt	#better
$n = 50$	RIH-PyramidalP	2.30	6.82	42
	RIH-Chains	2.24	0.89	11
	RIH-$S(3,1)$	0	0.43	50
	RIH-$S(2,1)$	0	1.01	50
$n = 100$	RIH-PyramidalP	2.56	6.04	46
	RIH-Chains	2.54	2.90	32
	RIH-$S(3,1)$	0	1.52	50
	RIH-$S(2,1)$	0.03	2.31	50
$n = 200$	RIH-PyramidalP	3.13	6.37	46
	RIH-Chains	3.09	4.76	40
	RIH-$S(3,1)$	0.02	2.29	50
	RIH-$S(2,1)$	0.03	2.92	50
$n = 500$	RIH-PyramidalP	3.18	5.94	49
	RIH-Chains	3.17	5.60	47

are similar to those of the third experiment. With the single-start, there is no domination relationship among the algorithms. When the experimental time is controlled, all the proposed algorithms (except RIH-Chains for $n = 50$) perform better than GRASP-2-opt. In addition, the results suggest that the larger the instance size, the greater the superiority of the proposed algorithms to GRASP-2-opt.

Table 3.8: Experimental results of the proposed algorithms and GRASP-2-opt with single-start on weighted-tree instances: average solution quality and running time for each instance set

	$n = 50$		$n = 100$		$n = 200$		$n = 500$	
	T	$G\%$	T	$G\%$	T	$G\%$	T	$G\%$
RIH-PyramidalP	0.0005	4.21	0.005	5.15	0.046	4.92	0.739	0.09
RIH-Chains	0.0479	3.54	0.135	4.84	0.360	4.66	1.684	0.01
RIH-$S(3,1)$	0.3023	0.62	3.175	0.47	34.141	0.38		
RIH-$S(2,1)$	0.0567	1.17	0.582	1.24	6.668	0.94		
GRASP-2-opt	0.0002	21.72	0.001	19.15	0.008	17.53	0.151	10.375

Table 3.9: Comparison of results of proposed algorithms and GRASP-2-opt on weighted-tree instances with controlled running time

		G% of proposed	G% of GRASP-2-opt	#better
$n = 50$	RIH-PyramidalP	4.21	11.16	46
	RIH-Chains	3.54	2.95	25
	RIH-$S(3,1)$	0.62	1.40	35
	RIH-$S(2,1)$	1.17	2.93	37
$n = 100$	RIH-PyramidalP	5.15	11.98	46
	RIH-Chains	4.84	7.03	37
	RIH-$S(3,1)$	0.47	4.31	46
	RIH-$S(2,1)$	1.24	5.50	46
$n = 200$	RIH-PyramidalP	4.92	12.27	50
	RIH-Chains	4.66	9.36	47
	RIH-$S(3,1)$	0.38	6.01	50
	RIH-$S(2,1)$	0.94	7.12	50
$n = 500$	RIH-PyramidalP	0.09	6.38	50
	RIH-Chains	0.01	5.64	50

3.5 Conclusions and Future Research

In this chapter, we provided the first ever detailed analyses and comparison of the effectiveness of various simple heuristics for the CTSPP by conducting extensive computational experiments. The results suggest that the proposed tour construction heuristics, IGRASP and RIH, perform better than the widely used GRASP, and 2-opt is the 'best' classical simple tour improvement heuristic for comparison with the proposed heuristics.

The proposed tour improvement heuristics, i.e., the pyramidal heuristic, chains heuristic and sliding window heuristic, are motivated by solvable cases for the CTSPP. These proposed heuristics have simple procedures and showed promising performance in the experiments. The proposed algorithms perform better than the classical GRASP-2-opt on general cases and weighted tress.

These analyses also give insight into the better selection and combination of simple heuristics in the future study of metaheuristics for the CTSPP. The outperformance of the proposed heuristics compared with the classical simple heuristics

indicates their potential for incorporation into future metaheuristics.

In the future, we aim to propose better metaheuristics based on the findings in this chapter. The proposed heuristics may also be applied to more general problems such as the load-dependent TSP and the cumulative VRP. Increasing the efficiency of the proposed algorithms for larger-size CTSPP is another future research topic.

Chapter 4

Heuristics for the Special CTSPP

4.1 Introduction and Related Works

In this chapter, the emphasis will be on the special cases and how solvable cases can be applied to heuristics for the CTSPP.

In the literature, polynomially solvable cases of the TSP have been intensively studied. We refer the readers to the well-known surveys of Lawler [104] and Burkard et al [27] for the solvable cases of the TSP. However, the research on solvability for the CTSPP has received insufficient attention. For the CTSPP, there exist algorithms for the specially structured cases including Line-CTSPP [3], edge-unweighted trees [111], three-diameter trees [20] and trees with a constant number of leaves [98], which can be solved in polynomial time.

One of our contributions is to enrich theoretical research on the CTSPP. First, we extend one solvable case, Line-CTSPP, to more general cases. Second, we find the relationship between the CTSPP and the quadratic assignment problem (QAP, [97]) and prove that the CTSPP on the SUM matrix can be solved within $O(nlogn)$ time. Third, we formulate the conjecture that the CTSPP on a subclass of convex-hull cases along two rays is NP-hard.

Firstly, Afrati et al. [3] solved Line-CTSPP, where all points are on a straight line, within $O(n^2)$ time using a dynamic programming algorithm. In classical theory, all nodes in the complete path are along a straight line, where the depot

is the first node and the exit node is not fixed. The theories of Line-CTSPP are extended by fixing the exit node and considering the straight line as part of the total path, where the depot on the line is no longer the first point in the complete path. We show that the extended cases can be solved with a dynamic programming algorithm in polynomial time. We also prove that the more general problem of Line-CTSPPW can be solved in polynomial time.

Secondly, the CTSPP can be transformed from the QAP, which was introduced by Koopmans and Beckmann [97], to model a plant location problem. The QAP is widely considered as a classical combinatorial optimisation problem and has been explored by mathematicians, computer scientists and operational researchers. More information on the QAP can be seen in the papers by Lawler [103, 102], Burkard [26], Rendl et al. [130] and Deineko and Woeginger [51]. The SUM matrix is a polynomially solvable case of the QAP and can be solved within $O(n^3)$ time [32]. In our research, the relationship between the QAP and the CTSPP is found, and the time complexity of the CTSPP on the SUM matrix is proven to be $O(nlogn)$.

Thirdly, Erickson [58] used a reduction argument to prove that a problem is NP-hard: to prove that problem B is NP-hard, a known NP-hard problem A is reduced to problem B. Reducing problem A to problem B involves finding an algorithm to solve problem A under the assumption that an algorithm for problem B already exists. The logic is essentially proof by contradiction. The reduction implies that if problem B was not hard, then there would exist an efficient algorithm to solve problem A, which is not the case. This indicates that if a special case A is a subclass of special case B and the CTSPP on A is NP-hard, then the CTSPP on B is also NP-hard. Çela et al. [35] suggested that given an edge-weighted tree, if the nodes are numbered a depth-first route, then the shortest path distance c_{ij} between nodes i and j determines a Kalmanson matrix. This means that the CTSPP on weighted trees can be reduced to the CTSPP on a Kalmanson matrix. Sitters [138] stated that the CTSPP is strongly NP-hard for weighted trees. Therefore, the CTSPP on a Kalmanson matrix is also NP-hard. However, it is not known whether the CTSPP on a convex hull is NP-hard. As reported in this chapter, we conduct extensive computational experiments on the convex-hull case and its subclass the two-ray case. The 'difficult' cases where the optimal solution cannot be obtained using the proposed heuristics allow us to formulate the conjecture that the CTSPP on two rays is NP-hard. This may provide the research community with an idea of proving NP-hardness in the future. Because we can reduce two-ray cases to convex-

hull cases, if the CTSPP could be proved to be NP-hard on two-ray cases, then the CTSPP on the convex hull would be NP-hard, which would be a significant theoretical contribution.

In the current research on specially structured cases for the CTSPP, most effort has been devoted to theoretical studies. Relevant heuristics based on solvable cases are scarce in the literature. This scarcity motivated the empirical research on the heuristics inspired by solvability in this chapter. In Chapter 3 we proposed heuristics motivated by solvable neighbourhoods and the solvable size, which perform well on general cases, while in this chapter we propose three dynamic programming heuristics motivated by the solvable specially structured matrices. These heuristics deliver a good performance for specially structured cases.

The first heuristic, the Line heuristic, is based on the solvable case of Line-CTSPP. The underlying matrix of a line distribution is an anti-Robinson matrix, as discussed in Chapter 2. This proposed heuristic outperforms the classical GRASP-2-opt in both running time and solution quality when the nodes are distributed close to a line following two parallel lines.

The second heuristic, the Up-Down heuristic, is inspired from one solvable case: the Path TSP on the convex hull. From Chapter 2, it is known that any convex-hull case satisfies the Kalmanson conditions. Garcia and Tejel [67] showed that the optimal Path TSP tour on the convex hull has a so-called Up-Down structure and can be found in polynomial time. In this chapter, we explain the special structure in more detail and propose a dynamic programming algorithm to find the optimal tour among the tours with this structure for the Path TSP. This special structure inspires the Up-Down heuristic, which can explore the best tour with this Up-Down structure for the CTSPP. We compare the Up-Down heuristic with the classical GRASP-2-opt on both convex-hull cases and close-to-convex-hull cases. When using the single-start, the proposed algorithm can obtain solutions of higher quality but with a longer running time. When the running time is controlled to be the same for both algorithms, the proposed heuristic outperforms GRASP-2-opt for all experimental sizes.

The third heuristic, the Two-Ray heuristic, combines the Up-Down structure discussed above with the Line structure on Line-CTSPP. The performance of this heuristic will be shown on a special case, where nodes are distributed along two rays.

The remainder of this chapter is structured as follows. In Section 4.2, we extend the theories of Line-CTSPP and propose the Line heuristic. In Section 4.3,

we prove the time complexity of the CTSPP on the SUM matrix. We consider two cases: a path with a fixed depot d and a path without a fixed depot. Section 4.4 demonstrates the UP-Down structure for the Path TSP on the convex hull and proposes the Up-Down heuristic and the Two-Ray heuristic for the CTSPP. In Section 4.5, we report computational experiments conducted to show the potential of the proposed heuristics on specially constructed cases. Section 4.6 concludes this chapter.

4.2 Extended Theory of Line-CTSPP

4.2.1 Line-CTSPP: Classical Simple Case

Line-CTSPP illustrated in Figure 4.1 is from the research of Afrati et al. [3]. In this figure, $d = x_0 = y_0$ is the depot, $x_1, \ldots, x_k$ are the nodes on the right of the depot, and $y_1, \ldots, y_m$ are the nodes on the left of the depot. There are n nodes in total with $m + k + 1 = n$. In the classical case, the depot is the entry point and the exit point is not fixed.

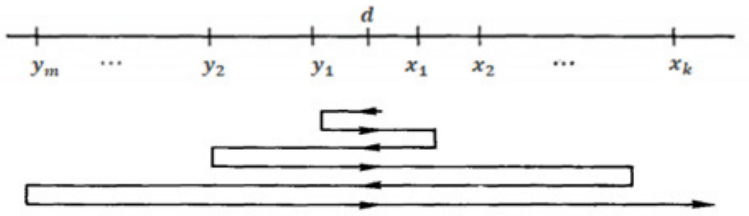


Figure 4.1: Illustration of the classical Line-CTSPP: node distribution and path structure

Lemma 4.2.1 *For Line-CTSPP, $x_i(y_i)$ should be visited earlier than $x_j(y_j)$ for all $1 \leq i < j \leq k(m)$ in the optimal tour.*

This lemma, proposed by Afrati et al. [3], suggests that when a node is passed by, it must be visited in the optimal tour. The proof of this lemma could not be found in the literature, so it is proven as follows:

Proof. W.l.o.g. we focus on the right side of the line. First, (x_j, x_i) is defined as a violated pair in the tour if x_j is visited earlier than x_i. P denotes the set of violated pairs in a tour τ. $(s_j, s_i) \in P$ is denoted as the first violated pair,

where s_j is the leftmost among all violated x_j, and s_i is the leftmost among x_i for all $(s_j, x_i) \in P$. A so-called tour improvement technique (TI-technique) is used to prove this lemma. The TI-technique is summarised in Algorithm 4.2.2 and was used by Burkard et al. [27] on solvable cases of the TSP. Assume that we start from an arbitrary tour τ with p violated pairs in total. A sequence of tours constructed is denoted as $\tau^1, \tau^2, \ldots, \tau^T$, with $\tau^1 = \tau$, such that $c(\tau^1) \geq c(\tau^2) \geq \cdots \geq c(\tau^T)$, where $c(\tau^t)$ denotes the length of tour $\tau^t (t = 1, \ldots, T)$ and $T = T(\tau)$ is the smallest integer such that τ^T is a tour satisfying the conditions described in this lemma. Tour τ^{t+1} is obtained from τ^t by removing one node x_i and inserting it in the previous position of x_j. Here, $(x_j, x_i) \in P$. This operation is called the Transformation. The Transformation is feasible if $c(\tau^{t+1}) \leq c(\tau^t)$ after the operation for all $t = 1, \ldots, T-1$. The feasibility of the Transformation follows from the illustration below.

Algorithm 4.2.2

TI-technique for Line-CTSPP with an unfixed exit

Input: a tour τ;
Output: a tour with x_i visited before x_j for all $1 \leq i < j \leq k$;
$\tau^1 \leftarrow \tau; t \leftarrow 1$; *$p \leftarrow$ number of violated pairs in τ;*
while $p \neq 0$ ***do***
  Find the first violated pair (s_j, s_i) in τ^t;
  Transformation: obtain τ^{t+1} from τ^t by removing s_i and inserting it before s_j;
  $t \leftarrow t + 1$;
  $p \leftarrow$ number of violated pairs in τ^t;
return τ^t.

Illustration: W.l.o.g. assume $\tau^t = \langle depot, \ldots, x, s_j, \ldots, y, s_i, z \ldots \rangle$, where x is on the a^{th} position and y is the b^{th} node in the tour. Figure 4.2 shows the positions of the first violated pair (s_j, s_i) in τ^t. Since (s_j, s_i) is the first violated pair, x should be on the left side of s_i on the line. Note that x in the figure is on the right of the depot, but x can also be the depot (i.e., $a = 1$), or a node on the left of the depot. After the Transformation, we have $\tau^{t+1} = \langle depot, \ldots, x, s_i, s_j, \ldots, y, z \ldots \rangle$. The cumulative costs of the two tours are given below:

$$\begin{aligned} c(\tau^t) &= c_t + (n-a) \times c_{x,s_j} + (n-b) \times c_{y,s_i} + (n-b-1) \times c_{s_i,z}; \\ c(\tau^{t+1}) &= c_{t+1} + (n-a) \times c_{x,s_i} + (n-a-1) \times c_{s_i,s_j} + (n-b-1) \times c_{y,z}. \end{aligned} \tag{4.1}$$

c_t includes the cost of partial tours $\langle depot, \ldots, x\rangle$, $\langle s_j, \ldots, y\rangle$ and $\langle z, \ldots\rangle$ in τ^t; c_{t+1} includes the cost of partial tours $\langle depot, \ldots, x\rangle$, $\langle s_j, \ldots, y\rangle$ and $\langle z, \ldots\rangle$ in τ^{t+1}. It can be seen that $c_{t+1} < c_t$ because the partial tour $\langle s_j, \ldots, y\rangle$ is moved backward in the new tour. Also, $(n-a) \times c_{x,s_i} + (n-a-1) \times c_{s_i,s_j} = (n-a-1) \times c_{x,s_j} + c_{x,s_i} < (n-a) \times c_{x,s_j}$; $(n-b) \times c_{y,s_i} + (n-b-1) \times c_{s_i,z} > (n-b-1) \times c_{y,z} + c_{y,s_i} > (n-b-1) \times c_{y,z}$. Therefore, $c(\tau^{t+1}) < c(\tau^t)$, hence the Transformation is feasible. This illustration can also be applied to the cases $\langle depot, \ldots, x, s_j, \ldots, y, s_i\rangle$, $\langle depot, \ldots, x, s_j, s_i, z \ldots\rangle$ and $\langle depot, \ldots, x, s_j, s_i\rangle$. It is easy to see that after each Transformation, the value of p is reduced by at least 1, so the TI-technique ends after at most p iterations. Thus the lemma is proved. □

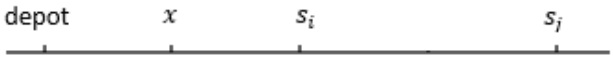


Figure 4.2: Illustration of positions of the first violated pair (s_j, s_i) in τ^t, which is used in the TI-technique on Line-CTSPP

Based on Lemma 4.2.1, the optimal objective value can be calculated with the dynamic programming recursions. Denote $R(y_j, x_i)$ as the minimal cost of the uncompleted path that has visited nodes $d, x_1, x_2, \ldots, x_i, y_1, y_2, \ldots, y_j$, and ends on node x_i. $L(y_j, x_i)$ has a similar definition, but the path of visited nodes ends on the left side on node y_j. Then the value $R(y_0, x_0)$ or $L(y_0, x_0)$ is the optimal objective value. The dynamic programming recursions are given below.

$$\begin{aligned} &R(y_m, x_k) = 0; \\ &L(y_m, x_k) = 0. \end{aligned} \tag{4.2}$$

$$\begin{aligned} &\text{When } j = m \text{ and } i = k-1, k-2, \ldots, 0: \\ &R(y_m, x_i) = R(y_m, x_{i+1}) + c(x_i, x_{i+1}) \times (n-i-m-1); \\ &L(y_m, x_i) = R(y_m, x_{i+1}) + c(y_m, x_{i+1}) \times (n-i-m-1). \end{aligned} \tag{4.3}$$

$$\begin{aligned} &\text{When } i = k \text{ and } j = m-1, m-2, \ldots, 0: \\ &R(y_j, x_k) = L(y_{j+1}, x_k) + c(x_k, y_{j+1}) \times (n-k-j-1); \\ &L(y_j, x_k) = L(y_{j+1}, x_k) + c(y_j, y_{j+1}) \times (n-k-j-1). \end{aligned} \tag{4.4}$$

When $j = m-1, m-2, \ldots, 0$ and $i = k-1, k-2, \ldots, 0$:

$$R(y_j, x_i) = min \begin{cases} R(y_j, x_{i+1}) + c(x_i, x_{i+1}) \times (n-i-j-1); \\ L(y_{j+1}, x_i) + c(x_i, y_{j+1}) \times (n-i-j-1). \end{cases} \tag{4.5}$$

$$L(y_j, x_i) = min \begin{cases} R(y_j, x_{i+1}) + c(y_j, x_{i+1}) \times (n-i-j-1); \\ L(y_{j+1}, x_i) + c(y_j, y_{j+1}) \times (n-i-j-1). \end{cases}$$

$$Opt = L(y_0, x_0) = R(y_0, x_0) = min \begin{cases} R(d, x_1) + c(d, x_1) \times (n-1); \\ L(y_1, d) + c(d, y_1) \times (n-1). \end{cases} \tag{4.6}$$

In the recursions, we demonstrate how to calculate $R(y_j, x_i)$, in which the sub-path has visited nodes $d, x_1, x_2, \ldots, x_i, y_1, y_2, \ldots, y_j$ and ended on x_i. When there are nodes remaining on both sides, i.e., when $j < m$ and $i < k$, based on Lemma 4.2.1, the next point visited can only be x_{i+1} or y_{j+1}. If the next node is x_{i+1}, $R(y_j, x_i)$ can be calculated from $R(y_j, x_{i+1})$; if the next node is y_{j+1}, $R(y_j, x_i)$ can be calculated from $L(y_{j+1}, x_i)$. In the recursions, $R(y_j, x_{i+1})$ and $L(y_{j+1}, x_i)$ have been calculated in the previous steps. The difference between $R(y_j, x_i)$ and $R(y_j, x_{i+1})$ is the cost of edge (x_i, x_{i+1}), while the difference between $R(y_j, x_i)$ and $L(y_{j+1}, x_i)$ is the cost of edge (x_i, y_{j+1}). Because x_i is the $(1+i+j)^{th}$ node in the path, the coefficient of edges (x_i, x_{i+1}) and (x_i, y_{j+1}) should be $n-1-i-j$. Using the minimisation criterion, we can determine the value of $R(y_j, x_i)$ and which node (x_{i+1} or y_{j+1}) is the next node. Similar calculations are applied to $L(y_j, x_i)$.

When all the nodes on one side have been visited, all the remaining nodes on the other side should be visited in order. This is shown in the recursions when $j = m$ or $i = k$. For example, $R(y_m, x_i)$ can be calculated when all nodes on the left side have been visited, ending on x_i. Because the next node can only be x_{i+1}, we can calculate $R(y_m, x_i)$ from $R(y_m, x_{i+1})$, which has been calculated in the previous steps. The difference between $R(y_m, x_i)$ and $R(y_m, x_{i+1})$ is the cost of edge (x_i, x_{i+1}). Because x_i is the $(1+i+m)^{th}$ node, the coefficient of edge (x_i, x_{i+1}) should be $n-1-i-m$, which is equal to $k-i$. The boundary conditions suggest that the minimal cost of the uncompleted path is 0 when all nodes are visited. The time complexity is $O(n^2)$.

4.2.2 Line-CTSPP: Fixed Exit Point

The previous section considered the Line-CTSPP with an unfixed exit node. Next, the Line-CTSPP with a fixed exit is considered. In other words, the last point visited on the line is now fixed. The objective is still to find the optimal tour on the line. This can be regarded as an (s, t) path problem.

The (s, t) path problem has been studied for the Path TSP, where the origin node s and destination node t are fixed [86]. The objective of the (s, t) Path TSP is the same as that of the classical Path TSP. The only difference is that the vehicle must return to a fixed point t. In the literature, Çela et al. [35] showed that the (s, t) Path TSP on Demidenko matrices can be solved in polynomial time.

In this section, we aim to extend the classical Line-CTSPP to the (s, t) Line-CTSPP, thus making a theoretical contribution. Assume that in Line-CTSPP, the vehicle departs from depot d and ends at the exit point e, i.e., the (d, e) path for Line-CTSPP. W.l.o.g. e is assumed to be on the right side of d, as illustrated in Figure 4.3. On this line, there are $n = m + k + 2$ nodes in total.

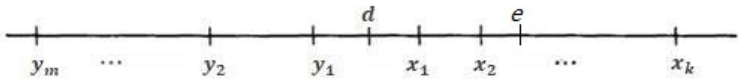


Figure 4.3: Illustration of Line-CTSPP with a fixed exit: distribution of nodes along a line

Lemma 4.2.3 *For Line-CTSPP with a fixed exit point e, $x_i(y_i)$ should be visited earlier than $x_j(y_j)$ for all $1 \leq i < j \leq k(m)$ in the optimal tour, and the last edge is (y_m, e) or (x_k, e) .*

Lemma 4.2.3 is similar to Lemma 4.2.1, in which there is no fixed exit. The TI-technique is also used to prove this lemma. W.l.o.g. we assume that $\tau^t = \langle depot, \ldots, x, s_j, \ldots, y, s_i, z \ldots, e \rangle$, where the positions of x and y are still a and b respectively in the tour, and e is the last node in the path. We still assume that (s_j, s_i) is the first violated pair, so that x is on the left side of s_i on the line. Here we consider four possible relative positions of the exit point e on the line as in Figure 4.4. After the Transformation, we have $\tau^{t+1} = \langle depot, \ldots, x, s_i, s_j, \ldots, y, z \ldots, e \rangle$. The cumulative costs of the two partial tours are calculated as follows:

$$\begin{aligned} c(\tau^t) &= c_t + (n-a) \times c_{x,s_j} + (n-b) \times c_{y,s_i} + (n-b-1) \times c_{s_i,z}; \\ c(\tau^{t+1}) &= c_{t+1} + (n-a) \times c_{x,s_i} + (n-a-1) \times c_{s_i,s_j} + (n-b-1) \times c_{y,z}. \end{aligned} \tag{4.7}$$

c_t includes the cost of partial tours $\langle depot, \ldots, x\rangle$, $\langle s_j, \ldots, y\rangle$ and $\langle z, \ldots, e\rangle$ in τ^t; c_{t+1} includes the cost of partial tours $\langle depot, \ldots, x\rangle$, $\langle s_j, \ldots, y\rangle$ and $\langle z, \ldots, e\rangle$ in τ^{t+1}. For all four cases in Figure 4.4, the proof is similar to that in Lemma 4.2.1: $c_{t+1} < c_t$ because the partial tour $\langle s_j, \ldots, y\rangle$ is moved backward in the new tour; $(n-a) \times c_{x,s_i} + (n-a-1) \times c_{s_i,s_j} = (n-a-1) \times c_{x,s_j} + c_{x,s_i} < (n-a) \times c_{x,s_j}$ and $(n-b) \times c_{y,s_i} + (n-b-1) \times c_{s_i,z} > (n-b-1) \times c_{y,z} + c_{y,s_i} > (n-b-1) \times c_{y,z}$. Therefore, $c(\tau^{t+1}) < c(\tau^t)$ is proved. This argument can also be applied to the other cases of τ^t: $\langle depot, \ldots, x, s_j, \ldots, y, s_i, e\rangle$, $\langle depot, \ldots, x, s_j, s_i, z, \ldots, e\rangle$ and $\langle depot, \ldots, x, s_j, s_i, e\rangle$. The lemma is proved.

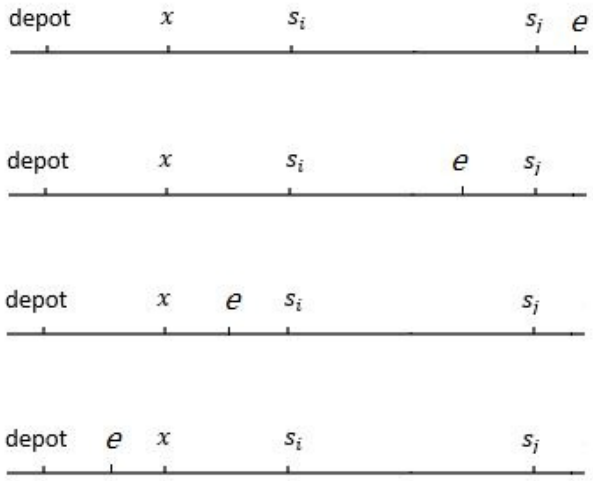


Figure 4.4: Illustration of four possible relative positions of the first violated pair (s_j, s_i) and exit point e in τ^t in the TI-technique on Line-CTSPP with a fixed exit

The same notations as in the previous section are used, and the dynamic programming recursions are as follows.

$$\begin{aligned} L(y_m, x_k) &= c(y_m, e); \\ R(y_m, x_k) &= c(x_k, e). \end{aligned} \tag{4.8}$$

$$\begin{aligned} &\text{When } j = m \text{ and } i = k-1, k-2, \ldots, 0: \\ &R(y_m, x_i) = R(y_m, x_{i+1}) + c(x_i, x_{i+1}) \times (n-i-m-1); \\ &L(y_m, x_i) = R(y_m, x_{i+1}) + c(y_m, x_{i+1}) \times (n-i-m-1). \end{aligned} \tag{4.9}$$

$$\begin{aligned}
&\text{When } i = k \text{ and } j = m-1, m-2, \ldots, 0: \\
&R(y_j, x_k) = L(y_{j+1}, x_k) + c(x_k, y_{j+1}) \times (n-k-j-1); \\
&L(y_j, x_k) = L(y_{j+1}, x_k) + c(y_j, y_{j+1}) \times (n-k-j-1).
\end{aligned} \tag{4.10}$$

$$\begin{aligned}
&\text{When } j = m-1, m-2, \ldots, 0 \text{ and } i = k-1, k-2, \ldots, 0: \\
&R(y_j, x_i) = min \begin{cases} R(y_j, x_{i+1}) + c(x_i, x_{i+1}) \times (n-i-j-1); \\ L(y_{j+1}, x_i) + c(x_i, y_{j+1}) \times (n-i-j-1). \end{cases} \\
&L(y_j, x_i) = min \begin{cases} R(y_j, x_{i+1}) + c(y_j, x_{i+1}) \times (n-i-j-1); \\ L(y_{j+1}, x_i) + c(y_j, y_{j+1}) \times (n-i-j-1). \end{cases}
\end{aligned} \tag{4.11}$$

$$Opt = L(y_0, x_0) = R(y_0, x_0) = min \begin{cases} R(d, x_1) + c(d, x_1) \times (n-1); \\ L(y_1, d) + c(d, y_1) \times (n-1). \end{cases} \tag{4.12}$$

The recursions are the same as in the previous section, except that the boundary conditions of $L(y_m, x_k)$ and $R(y_m, x_k)$ change their values to the cost of the last edge (y_m, e) and (x_k, e) respectively.

4.2.3 Line-CTSPP: Part CTSPP Line with a Fixed Exit

In Section 4.2.1 and 4.2.2, all the nodes in the complete path were along a straight line and the depot was always the first node. This problem can be seen as the 'Pure CTSPP Line'. We now consider the Part CTSPP Line, in which the straight line is only a part of the total path. In general, only some of the nodes are located along the straight line. The depot d is the first node visited in the line part, instead of the first node visited in the complete path. We consider this problem as this case will be applied to the Two-Ray heuristic later. An instance of this extended case is illustrated in Figure 4.5. There are n nodes in the complete path and $m+k+2$ nodes on the straight line. Here $n \geq m+k+2$. When $n = m+k+2$, the complete path consists of all the nodes on the line, which is the case in Section 4.2.2.

For the Part CTSPP Line, it is assumed that some of the nodes that are

not on the line are visited first, followed by depot d, and then all nodes on the line. The last visited node on the line is still the exit node e as in Section 4.2.2, and finally the remaining nodes are visited. In this case, we still focus on the line part and aim to find the optimal sub-tour that minimises the sum of waiting times for all nodes on the line. Assume that depot d is the p^{th} node in the complete path. In the Pure CTSPP Line, $p = 1$, and the coefficients of the costs of edges visited on the line are $n-1, n-2, \ldots, 1$ in consecutive order, while in the Part CTSPP Line, the coefficients are changed to $n-p, n-(p+1), \ldots, n-(p+m+k)$. It can be proved that Lemma 4.2.3 remains valid for the partial path with nodes on the line.

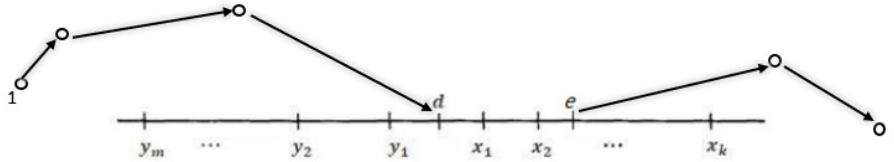


Figure 4.5: Illustration of the Part CTSPP Line with a fixed exit: distribution of nodes along a line

The proof is similar to that in Section 4.2.2. The TI-technique is again used to illustrate the proof, and we only focus on the line part. We still assume that the partial path on the line is $\tau^t = \langle depot, \ldots, x, s_j, \ldots, y, s_i, z \ldots \rangle$. From Section 4.2.2, it is known that x and y are the a^{th} and b^{th} nodes in the complete path respectively. However, for the Part CTSPP Line, the position of x in the complete path (including the line part and non-line part) is a^* instead of a, where $a^* = p + a - 1$. Similarly, y is the $(b^*)^{th}$ node in the complete path, where $b^* = p + b - 1$. Therefore, we only need to replace a and b in the proof in Section 4.2.2 with a^* and b^* respectively.

We can now use the following dynamic programming recursions to find the optimal sub-tour that can minimise the sum of waiting times for all nodes on the line. The notations are the same as before.

$$\begin{aligned} L(y_m, x_k) &= c(y_m, e) \times (n-k-m-p); \\ R(y_m, x_k) &= c(x_k, e) \times (n-k-m-p). \end{aligned} \tag{4.13}$$

When $j = m$ and $i = k-1, k-2, \ldots, 0$:

$$\begin{aligned} R(y_m, x_i) &= R(y_m, x_{i+1}) + c(x_i, x_{i+1}) \times (n - i - m - p); \\ L(y_m, x_i) &= R(y_m, x_{i+1}) + c(y_m, x_{i+1}) \times (n - i - m - p). \end{aligned} \tag{4.14}$$

When $i = k$ and $j = m-1, m-2, \ldots, 0$:

$$\begin{aligned} R(y_j, x_k) &= L(y_{j+1}, x_k) + c(x_k, y_{j+1}) \times (n - k - j - p); \\ L(y_j, x_k) &= L(y_{j+1}, x_k) + c(y_j, y_{j+1}) \times (n - k - j - p). \end{aligned} \tag{4.15}$$

When $j = m-1, m-2, \ldots, 0$ and $i = k-1, k-2, \ldots, 0$:

$$\begin{aligned} R(y_j, x_i) &= min \begin{cases} R(y_j, x_{i+1}) + c(x_i, x_{i+1}) \times (n - i - j - p); \\ L(y_{j+1}, x_i) + c(x_i, y_{j+1}) \times (n - i - j - p). \end{cases} \\ L(y_j, x_i) &= min \begin{cases} R(y_j, x_{i+1}) + c(y_j, x_{i+1}) \times (n - i - j - p); \\ L(y_{j+1}, x_i) + c(y_j, y_{j+1}) \times (n - i - j - p). \end{cases} \end{aligned} \tag{4.16}$$

$$Opt = L(y_0, x_0) = R(y_0, x_0) = min \begin{cases} R(d, x_1) + c(d, x_1) \times (n - p); \\ L(y_1, d) + c(d, y_1) \times (n - p). \end{cases} \tag{4.17}$$

The recursions are similar to those in Section 4.2.2. The only difference is the coefficients. Because the depot is the p^{th} point in the complete path, the positions of all nodes on the line should start from p. For example, when calculating $R(y_j, x_i)$, the sub-path on the line part has visited $d, x_1, x_2, \ldots, x_i, y_1, y_2, \ldots, y_j$ and ended on x_i. Based on Lemma 4.2.3, the next point visited can only be x_{i+1} or y_{j+1}. Therefore, the position of x_i in the complete path is $i + j + p$; thus, the coefficient of edge (x_i, x_{i+1}) and edge (x_i, y_{j+1}) should be $n - i - j - p$. Note that when $p = 1$ and $n = m + k + 2$, the recursions here are exactly the same as those in Section 4.2.2.

4.2.4 Line-CTSPPW

In this section, we prove that a more general problem, CTSPP with weights (CTSPPW) on a line, can be solved in polynomial time.

The definition of the CTSPPW is similar to that of the CTSPP in Chapter 3. $V = \{v_1, \ldots, v_n\}$ is a vertex set, in which each vertex represents the location of one customer, and v_1 represents the depot. For each edge $(v_i, v_j) \in E$, $c_{v_i v_j}$ is the travelling time between v_i and v_j. For the CTSPPW, w_{v_i} is defined as the weight of node v_i. The objective of the CTSPPW is to find a Hamiltonian path starting from v_1 that minimises the sum of arrival times at all nodes with weights. We define a tour as $\tau = \langle \tau_1, \tau_2, \tau_3, \ldots, \tau_{n-1}, \tau_n \rangle$, where τ_i is the i^{th} vertex in a solution. Thus, the objective value can be written as:

$$c(\tau) = \sum_{i=1}^{i=n-1} (c_{\tau_i \tau_{i+1}} \times \sum_{j=i+1}^{j=n} w_{\tau_j}) \tag{4.18}$$

The node distribution for Line-CTSPPW is the same as for Line-CTSPP, which is illustrated in Figure 4.1.

Lemma 4.2.4 *For the CTSPPW on a line, $x_i(y_i)$ should be visited earlier than $x_j(y_j)$ for all $1 \leq i < j \leq k(m)$ in the optimal tour.*

The proof is similar to that in Section 4.2.1. We again use the TI-technique for the partial paths $\tau^t = \langle depot, \ldots, x, s_j, \ldots, y, s_i, z \ldots \rangle$ and $\tau^{t+1} = \langle depot, \ldots, x, s_i, s_j, \ldots, y, z \ldots \rangle$ to illustrate the proof. The positions of x and y are a and b respectively. The difference here is that the weights are added when the objective values are calculated. The cumulative costs of the two tours are given below.

$$\begin{aligned} c(\tau^t) &= c_t + (c_{x,s_j} \times \sum_{q=a+1}^{q=n} w_{\tau_q^t}) + (c_{y,s_i} \times \sum_{q=b+1}^{q=n} w_{\tau_q^t}) + (c_{s_i,z} \times \sum_{q=b+2}^{q=n} w_{\tau_q^t}); \\ c(\tau^{t+1}) &= c_{t+1} + (c_{x,s_i} \times \sum_{q=a+1}^{q=n} w_{\tau_q^{(t+1)}}) + (c_{s_i,s_j} \times \sum_{q=a+2}^{q=n} w_{\tau_q^{(t+1)}}) + (c_{y,z} \times \sum_{q=b+2}^{q=n} w_{\tau_q^{(t+1)}}). \end{aligned} \tag{4.19}$$

Similarly, c_t includes the cost of partial tours $\langle depot, \ldots, x \rangle$, $\langle s_j, \ldots, y \rangle$ and $\langle z, \ldots \rangle$ in τ^t; c_{t+1} includes the cost of partial tours $\langle depot, \ldots, x \rangle$, $\langle s_j, \ldots, y \rangle$ and $\langle z, \ldots \rangle$ in τ^{t+1}. $c_{t+1} < c_t$ because the partial tour $\langle s_j, \ldots, y \rangle$ is moved backward in the new

tour. $(c_{x,s_i} \times \sum_{q=a+1}^{q=n} w_{\tau_q^{(t+1)}}) + (c_{s_i,s_j} \times \sum_{q=a+2}^{q=n} w_{\tau_q^{(t+1)}}) = (c_{x,s_j} \times \sum_{q=a+2}^{q=n} w_{\tau_q^t}) + c_{x,s_i} \times w_{s_j} < (c_{x,s_j} \times \sum_{q=a+1}^{q=n} w_{\tau_q^t})$; $(c_{y,s_i} \times \sum_{q=b+1}^{q=n} w_{\tau_q^t}) + (c_{s_i,z} \times \sum_{q=b+2}^{q=n} w_{\tau_q^t}) = ((c_{y,s_i} + c_{s_i,z}) \times \sum_{q=b+2}^{q=n} w_{\tau_q^t}) + c_{y,s_i} \times w_{s_i} > (c_{y,z} \times \sum_{q=b+2}^{q=n} w_{\tau_q^{(t+1)}}) + c_{y,s_i} \times w_{s_i} > (c_{y,z} \times \sum_{q=b+2}^{q=n} w_{\tau_q^{(t+1)}})$. Therefore, $c(\tau^{t+1}) < c(\tau^t)$. This argument can also be applied to other cases: $\langle depot, \ldots, x, s_j, \ldots, y, s_i \rangle$, $\langle depot, \ldots, x, s_j, s_i, z \ldots \rangle$ and $\langle depot, \ldots, x, s_j, s_i \rangle$. The lemma is thus proved.

It has been proved that the added weights have no effect on the validity of Lemma 4.2.1. Therefore, we use the same notations as in previous sections and also define W_T, W_{TX} and W_{TY} as the total weight of all nodes, the total weight of nodes on the right side and the total weight of nodes on the left side respectively. The dynamic programming recursions are as follows, where the time complexity is $O(n^3)$.

$$\begin{aligned} R(y_m, x_k) &= 0; \\ L(y_m, x_k) &= 0. \end{aligned} \tag{4.20}$$

When $j = m$ and $i = k-1, k-2, \ldots, 0$:

$$\begin{aligned} R(y_m, x_i) &= R(y_m, x_{i+1}) + c(x_i, x_{i+1}) \times (W_{TX} - \sum_{q=1}^{q=i} w_{x_q}); \\ L(y_m, x_i) &= R(y_m, x_{i+1}) + c(y_m, x_{i+1}) \times (W_{TX} - \sum_{q=1}^{q=i} w_{x_q}). \end{aligned} \tag{4.21}$$

When $i = k$ and $j = m-1, m-2, \ldots, 0$:

$$\begin{aligned} R(y_j, x_k) &= L(y_{j+1}, x_k) + c(x_k, y_{j+1}) \times (W_{TY} - \sum_{q=1}^{q=j} w_{y_q}); \\ L(y_j, x_k) &= L(y_{j+1}, x_k) + c(y_j, y_{j+1}) \times (W_{TY} - \sum_{q=1}^{q=j} w_{y_q}). \end{aligned} \tag{4.22}$$

When $j = m-1, m-2, \ldots, 0$ and $i = k-1, k-2, \ldots, 0$:

$$R(y_j, x_i) = min \begin{cases} R(y_j, x_{i+1}) + c(x_i, x_{i+1}) \times (W_T - \sum_{q=1}^{q=i} w_{x_q} - \sum_{q=1}^{q=j} w_{y_q}); \\ L(y_{j+1}, x_i) + c(x_i, y_{j+1}) \times (W_T - \sum_{q=1}^{q=i} w_{x_q} - \sum_{q=1}^{q=j} w_{y_q}). \end{cases}$$

$$L(y_j, x_i) = min \begin{cases} R(y_j, x_{i+1}) + c(y_j, x_{i+1}) \times (W_T - \sum_{q=1}^{q=i} w_{x_q} - \sum_{q=1}^{q=j} w_{y_q}); \\ L(y_{j+1}, x_i) + c(y_j, y_{j+1}) \times (W_T - \sum_{q=1}^{q=i} w_{x_q} - \sum_{q=1}^{q=j} w_{y_q}). \end{cases} \tag{4.23}$$

$$Opt = L(y_0, x_0) = R(y_0, x_0) = min \begin{cases} R(d, x_1) + c(d, x_1) \times W_T; \\ L(y_1, d) + c(d, y_1) \times W_T. \end{cases} \tag{4.24}$$

4.2.5 Line Heuristic

Inspired by the solvable Line-CTSPP, the so-called Line heuristic is proposed. This heuristic is expected to deliver a good performance on cases where the distribution of nodes follows the close-to-line structure, a special structure illustrated in the experiments in Section 4.5.

Given a distance matrix C and a depot d, where the nodes are not distributed along a line, the first step is to number the nodes as $\tau = \langle y_m, \ldots, y_j, y_2, y_1, d, x_1, x_2, \ldots, x_i, x_k \rangle$ as in Line-CTSPP. We can see that such numbering describes a similar tour to that obtained from the DENN algorithm, which adds nodes on two sides (left or right) to update the two endpoints of the partial tour. Therefore, a tour with this numbering can be achieved with DENN. Then we apply the dynamic programming recursions in Section 4.2.1 to obtain a heuristic solution.

As discussed in Chapter 2, DENN uses the nearest neighbour rule to construct the tour. In DENN, a permutation starts from $\langle y_1, d \rangle$, where d is the depot, and y_1 is the nearest node to the depot. y_1 is referred to as the *left* node and d is referred to as the *right* node. The tour is then extended by adding the nearest node (which is the nearest node to the left or to the right) and updating the reference to the *left* and *right*. Recursively, given the partial tour $\langle y_j, \ldots, y_2, y_1, d, x_1, x_2, \ldots, x_i \rangle$, from the nodes not yet chosen, we assume that

a and b are the nearest neighbours to x_i and y_j respectively. We then compare the two lengths c_{a,x_i} and c_{b,y_j}, add the shorter edge to the corresponding side and update the corresponding endpoint. For example, if $c_{a,x_i} < c_{b,y_j}$, then a is added on the right side to obtain the partial tour $\langle y_j, \ldots, y_2, y_1, d, x_1, x_2, \ldots, x_i, a\rangle$. The process continues until all nodes are included in the permutation.

After the process of DENN, the path can be recorded as $\tau = \langle y_m, \ldots, y_j, y_2, y_1, d, x_1, x_2, \ldots, x_i, x_k\rangle$. Then, the dynamic programming recursions in Section 4.2.1 are used to find the heuristic solution.

4.3 CTSPP on SUM Matrix

4.3.1 Theoretical Result of CTSPP on SUM Matrix

The CTSPP can be transformed from the QAP, which can be demonstrated mathematically using the Koopmans-Beckmann form in [97]. Given the set $\{1, 2, \ldots, n\}$ and two $n \times n$ matrices $C = (c_{ij})$ and $B = (b_{ij})$, the QAP is denoted as QAP(C,B) with the following objective, where S_n is the set of permutations of $\{1, 2, \ldots, n\}$:

$$min_{\tau \in S_n} \sum_{i=1}^{n} \sum_{j=1}^{n} c_{\tau_i,\tau_j} \times b_{ij} \tag{4.25}$$

The value of this objective depends on the matrices C and B and the permutation τ. Now, we consider a polynomially solvable case of the QAP. A matrix C is called a SUM matrix [32] if there are two vectors $u = (u_1, u_2, \ldots, u_n)$ and $v = (v_1, v_2, \ldots, v_n)$ such that $c_{ij} = u_i + v_j$. In the literature, QAP(C,B) with an $n \times n$ SUM matrix C can be simplified to a linear assignment problem that can be solved in $O(n^3)$ time [32].

We find that there is a close relationship between CTSPP(C) and QAP(C,B). When matrix B has the special structure, in which $b_{12} = n-1, b_{23} = n-2, \ldots, b_{n-1,n} = 1$ and all other elements are 0, then QAP(C,B) is transformed into CTSPP(C). Here, matrix B can be considered as the coefficient matrix for the CTSPP.

Theorem 4.3.1 *For CTSPP(C), if the distance matrix C is the SUM matrix with $c_{ij} = u_i + v_j$, then the problem can be solved within $O(nlogn)$ time.*

Proof. Define a tour as $\tau = \langle \tau_1, \tau_2, \tau_3, \ldots, \tau_{n-1}, \tau_n\rangle$. Then, the objective value can be written as:

$$
\begin{aligned}
c(\tau) &= \sum_{i=1}^{n-1} c_{\tau_i \tau_{i+1}} \times (n-i) \\
&= \sum_{i=1}^{n-1} (u_{\tau_i} + v_{\tau_{i+1}}) \times (n-i) \\
&= n \sum_{i=1}^{n} u_{\tau_i} + (n+1) \sum_{i=1}^{n} v_{\tau_i} - (\sum_{i=2}^{n} (u_{\tau_i} + v_{\tau_i}) \times i + u_{\tau_1} + (n+1) v_{\tau_1})
\end{aligned} \tag{4.26}
$$

Let $\sum_{i=1}^{n} u_i$, $\sum_{i=1}^{n} v_i$ be sum_u and sum_v respectively. Then the first two items in formula 4.26 are constant values. To minimise the objective value $c(\tau)$, we should maximise the value of $\sum_{i=2}^{n}(u_{\tau_i} + v_{\tau_i}) \times i + u_{\tau_1} + (n+1)v_{\tau_1}$. From the inequalities theorem [83], we know that to maximise the scalar product of two vectors, the components of the two vectors should be sorted in the same direction. Mathematically, given two vectors $x = (x_1, x_2, \ldots, x_n)$, $y = (y_1, y_2, \ldots, y_n)$, $\sum_{i=1}^{n}(x_{\pi_i} \times y_{\tau_i})$ for all $\tau, \pi \in S_n$ is maximised when $x_{\pi_1} \leq x_{\pi_2} \leq \cdots \leq x_{\pi_n}$and $y_{\tau_1} \leq y_{\tau_2} \leq \cdots \leq y_{\tau_n}$, where S_n is the set of permutations of $\{1, 2, \ldots, n\}$. Based on the inequalities theorem, one can view $u + v$ as the vector x and $(1, 2, 3 \ldots, n)$ as the vector y. Therefore, we first calculate the value $u_i + v_i$ for all i and sort the summed values from the smallest to the largest to obtain the sorted list $sum = \langle u_{s_1} + v_{s_1}, u_{s_2} + v_{s_2}, \ldots, u_{s_n} + v_{s_n} \rangle$, which requires $O(nlogn)$ time. In this sorting process, the original index (position) s_i for all i can also be determined, which will be used in the tour construction process. Then, we set $\tau_1 = s_k (k = 1, 2, 3, \ldots, n)$, and calculate the maximum value of $\sum_{i=2}^{n}(u_{\tau_i} + v_{\tau_i}) \times i$ as $maxv_k$ for each k using the following recursions:

$$
\begin{aligned}
&\tau_1 = s_1, maxv_1 = (u_{s_2} + v_{s_2}) \times 2 + (u_{s_3} + v_{s_3}) \times 3 + \cdots + (u_{s_n} + v_{s_n}) \times n; \\
&\tau_1 = s_k, maxv_k = maxv_{k-1} - (u_{s_k} + v_{s_k}) \times k + (u_{s_{k-1}} + v_{s_{k-1}}) \times k \text{ for all } k = 2, \ldots, n.
\end{aligned} \tag{4.27}
$$

We then calculate and compare n values, $value_k = maxv_k + u_{s_k} + (n+1)v_{s_k}$ for $k = 1, 2, \ldots, n$, and choose the largest value as $value^*$. In this way, the optimal value can be found, which is $nsum_u + (n+1)sum_v - value^*$. In the process, when k is increased by 1, the calculation of $maxv_{k+1}$ is based on $maxv_k$, which only requires linear time $O(1)$. Thus, calculating n values $value_k$ requires $O(n)$ time.

Therefore, the time complexity of finding the optimal value is $O(nlogn)$.

Then the optimal tour can be constructed. When $\tau_1 = s_1$, the corresponding tour is $\langle s_1, s_2, \ldots, s_n \rangle$; when $\tau_1 = s_k$, the corresponding tour is constructed by removing s_k and placing it at the beginning of the tour, i.e., $\langle s_k, s_1, s_2, \ldots, s_{k-1}, s_{k+1}, \ldots, s_n \rangle$. The operation requires linear time $O(1)$. The largest $value_k$ is recorded as $value^*$, and the corresponding index k is recorded as k^*; thus, the optimal tour can be found as $\langle s_{k^*}, s_1, s_2, \ldots, s_{k^*-1}, s_{k^*+1}, \ldots, s_n \rangle$. Therefore, the time complexity of finding the optimal solution is still $O(nlogn)$. This completes the proof. □

A simple example is used to illustrate the procedure. Assume that $u = (3, 5, 1, 4, 4), v = (1, 4, 2, 2, 3)$, then $u + v = (4, 9, 3, 6, 7)$, $sum_u = 17, sum_v = 12$. We sort the elements in $u+v$ and obtain the sorted list $sum = \langle u_3+v_3, v_1+v_1, u_4+v_4, u_5+v_5, u_2+v_2 \rangle = \langle 3, 4, 6, 7, 9 \rangle$. Therefore, $s_1 = 3, s_2 = 1, s_3 = 4, s_4 = 5$, and $s_5 = 2$. $maxv_k$ for each k is calculated as follows.

$$\tau_1 = s_1 = 3, maxv_1 = 4 \times 2 + \cdots + 9 \times 5 = 99, value_1 = 112, \tau = \langle 3, 1, 4, 5, 2 \rangle,$$
$$\tau_1 = s_2 = 1, maxv_2 = 99 - 4 \times 2 + 3 \times 2 = 97, value_2 = 106, \tau = \langle 1, 3, 4, 5, 2 \rangle,$$
$$\tau_1 = s_3 = 4, maxv_3 = 97 - 6 \times 3 + 4 \times 3 = 91, value_3 = 107, \tau = \langle 4, 3, 1, 5, 2 \rangle,$$
$$\tau_1 = s_4 = 5, maxv_4 = 91 - 7 \times 4 + 6 \times 4 = 87, value_4 = 109, \tau = \langle 5, 3, 1, 4, 2 \rangle,$$
$$\tau_1 = s_5 = 2, maxv_5 = 87 - 9 \times 5 + 7 \times 5 = 77, value_5 = 106, \tau = \langle 2, 3, 1, 4, 5 \rangle.$$

We find that $value_1$ is the largest among the five values; thus, $k^* = 1$, $value^* = 112$. Therefore, the optimal value is $5 \times 17 + 6 \times 12 - 112 = 45$ with the optimal tour $\tau = \langle 3, 1, 4, 5, 2 \rangle$.

In this section, it is found that CTSPP(C) is a special case of QAP(C,B) with a special matrix B. In the literature, QAP(C,B) on the SUM matrix C can be solved within $O(n^3)$ time. This research enriches the theory on solvable cases by proving that the CTSPP on the SUM matrix can be solved within $O(nlogn)$ time.

4.3.2 Two Cases of CTSPP on SUM Matrix

We consider the symmetric distance matrix C with $c_{ij} = u_i + u_j$ in this section. A tour is still defined as $\tau = \langle \tau_1, \tau_2, \tau_3, \ldots, \tau_{n-1}, \tau_n \rangle$, then the objective value is given below.

$$c(\tau) = (2n+1)sum_u - (2\sum_{i=2}^{n} u_{\tau_i} \times i + (n+2)u_{\tau_1}) \tag{4.28}$$

Two cases are considered. In the first case, a path with a fixed depot d, i.e., $\tau_1 = d$ is constructed. In the second case, a path without a fixed depot, i.e., τ_1 can be any node, is constructed. The time complexity will be shown to be $O(nlogn)$ for both cases.

First case with fixed depot

W.l.o.g. we assume that $\tau_1 = d = 1$. One can maximise $\sum_{i=2}^{n} u_{\tau_i} \times i$ to obtain the optimal solution.

Based on the inequalities theorem, $u_2, u_3, \ldots, u_n$ is sorted from the smallest to the largest to obtain the sorted list $u' = \langle u'_1, u'_2, \ldots, u'_{n-1} \rangle = \langle u_{s_1}, u_{s_2}, \ldots, u_{s_{n-1}} \rangle$, which requires $O(nlogn)$ time. Through this sorting process, we can also determine the original index (position) $s_i \neq d$, which will be used in the tour construction process. Then, formula 4.29 is used to obtain the optimal value. The optimal tour is $\langle d, s_1, s_2, \ldots, s_{n-1} \rangle$ and the time complexity is $O(nlogn)$.

$$\begin{aligned} sum_u &= \sum_{i=1}^{n} u_i; \\ maxv^* &= \sum_{i=2}^{n} u'_{i-1} \times i; \\ opt &= (2n+1)sum_u - 2maxv^* - (n+2)u_d. \end{aligned} \tag{4.29}$$

A simple example is used to illustrate the procedure. Assume that $u = (3, 5, 1, 4, 4)$, then $sum_u = 17$. We sort $u_2, u_3, \ldots, u_5$ and obtain the sorted list $u' = \langle u'_1, u'_2, u'_3, u'_4 \rangle = \langle u_3, u_5, u_4, u_2 \rangle = \langle 1, 4, 4, 5 \rangle$. Then $maxv^* = \sum_{i=2}^{n} u'_{i-1} \times i = 55$, the optimal value is $(2 \times 5 + 1) \times 17 - 2 \times 55 - 7 \times 3 = 56$ and the optimal tour is $\langle 1, 3, 5, 4, 2 \rangle$.

Second case without fixed depot

We now consider the problem of the path without a fixed depot. From formula 4.28, it can be seen that the optimal solution is obtained by maximising $2\sum_{i=2}^{n} u_{\tau_i} \times i + (n+2)u_{\tau_1}$.

Similarly, we sort all $u_i, i = 1, 2, \ldots, n$ from the smallest to the largest and obtain the sorted list $u' = \langle u'_1, u'_2, \ldots, u'_n \rangle = \langle u_{s_1}, u_{s_2}, \ldots, u_{s_n} \rangle$, which requires $O(nlogn)$ time. In this sorting process, the index (position) s_i in the original vector

u is also recorded, which will be used in the tour construction process.

Then, $\tau_1 = s_k (k = 1, 2, 3, \ldots, n)$ is set, and $maxv_k$, which is the maximum value of $2\sum_{i=2}^{n} u_{\tau_i} \times i$, and $value_k$ for each k are calculated using the following recursions.

$$\begin{aligned}
&\tau_1 = s_1, maxv_1 = 2 \times (u'_2 \times 2 + u'_3 \times 3 + \cdots + u'_n \times n);\\
&\tau_1 = s_k, maxv_k = maxv_{k-1} + 2 \times (u'_{k-1} - u'_k) \times k \text{ for all } k = 2, \ldots, n;\\
&value_k = maxv_k + (n+2)u_{s_k} \text{ for all } k = 1, 2, \ldots, n.
\end{aligned} \tag{4.30}$$

The largest $value_k$ is recorded as $value^*$, then the optimal value is $opt = (2n+1)sum_u - value^*$.

In the process, when k is increased by 1, the calculation of $maxv_{k+1}$ is based on $maxv_k$, which only requires linear time $O(1)$; thus, calculating n values $value_k$ requires $O(n)$ time. Therefore, the time complexity of finding the optimal value is $O(nlogn)$. Then we construct the optimal tour. When $\tau_1 = s_1$, the corresponding tour is $\langle s_1, s_2, \ldots, s_n \rangle$; when $\tau_1 = s_k$, the corresponding tour is constructed by removing s_k and placing it at the beginning of the tour, i.e., $\langle s_k, s_1, s_2, \ldots, s_{k-1}, s_{k+1}, \ldots, s_n \rangle$. The operation requires linear time $O(1)$. The largest $value_k$ is recorded as $value^*$, and the corresponding index k is recorded as k^*; thus, the optimal tour is $\langle s_{k^*}, s_1, s_2, \ldots, s_{k^*-1}, s_{k^*+1}, \ldots, s_n \rangle$. Therefore, the time complexity of finding the optimal solution is still $O(nlogn)$.

4.4 CTSPP Heuristics Based on the Convex Hull

In this section, the Up-Down heuristic is proposed for the CTSPP, which can be applied to the convex-hull cases and close-to-convex-hull cases. Further, the Two-Ray heuristic is proposed for a special case of convex hull, where all nodes are along two rays.

First we demonstrate a solvable case: Path TSP on the convex hull, where the optimal solution has a special Up-Down structure [67]. This structure will be further used in the Up-Down heuristic and Two-Ray heuristic for the CTSPP.

4.4.1 Path TSP on Convex Hull

The node distribution along the boundary of a convex hull is demonstrated in Figure 4.6. Denote the set of nodes $U^i = \{2, 3, \ldots, i-2, i-1\}$ as the upper set of

node i, numbered clockwise; denote the set of nodes $L^i = \{n, n-1, \ldots, i+2, i+1\}$ as the lower set of i, numbered anticlockwise. Node 1 is the depot.

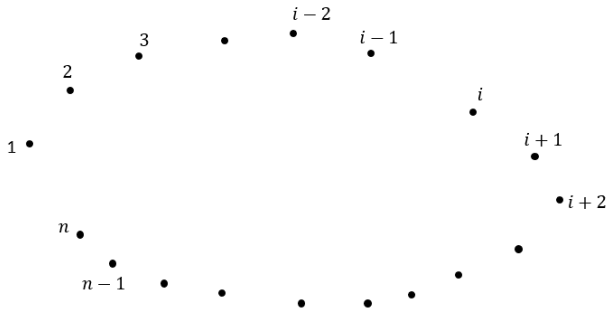


Figure 4.6: Illustration of the node distribution along a convex hull

Lemma 4.4.1 *For the Path TSP, the shortest path that visits nodes* $1, 2, \ldots, i$ *is path* $\langle 1, 2, 3, \ldots, i-2, i-1, i \rangle$.

Lemma 4.4.2 *In the optimal path of the Path TSP, the nodes in the sub-path from the depot to node* i *are visited in the same order as in* U^i *and* L^i.

The two lemmas are extracted from Garcia and Tejel's research [67] and were proved by the quadrangle inequality: the sum of the lengths of the diagonals is greater than the sum of the lengths of the two opposite sides. The quadrangle inequality suggests that the optimal path cannot intersect itself and follows the Up-Down structure. A detailed proof can be seen in [67].

We illustrate the Up-Down structure here. In the optimal path, assume that the last node visited is i, then all nodes in $U^i = \{2, 3, \ldots, a, \ldots, b, \ldots, i-2, i-1\}$ and $L^i = \{n, n-1, \ldots, c, \ldots, d, \ldots, i+2, i+1\}$ are visited in this order. In other words, b cannot be visited earlier than a, and d cannot be visited earlier than c. In addition, not all the nodes in U^i and L^i have to be visited at a single time. After departing from the depot, the nodes visited can be in $U^i, L^i, U^i, L^i, \ldots$ or $L^i, U^i, L^i, U^i, \ldots$ until node i is visited. In summary, in the optimal path, upper sets alternate with lower sets, and all nodes in the sets are visited in order.

Note that if the last node is $i = 2$, then the path following the Up-Down structure can only be $\langle 1, n, n-1, \ldots, 3, 2 \rangle$, in which case only the nodes in L^i are visited. Similarly, if the last node is $i = n$, then the path following the Up-Down structure can only be $\langle 1, 2, 3, \ldots, n-1, n \rangle$, in which case only the nodes in U^i are visited.

We use a simple example to explain this structure. Assume that $n = 10$ and that $i = 6$ is the last node in the optimal path. Then $U^6 = \{2, 3, 4, 5\}$ and $L^6 = \{10, 9, 8, 7\}$. The path $\langle 1, 2, 3, 10, 9, 8, 4, 5, 7, 6\rangle$ follows the Up-Down structure, while the path $\langle 1, 2, 4, 10, 9, 5, 8, 3, 7, 6\rangle$ violates the Up-Down structure as the nodes in U^6 are visited in the order $\langle 2, 4, 5, 3\rangle$ rather than $\langle 2, 3, 4, 5\rangle$.

Since the optimal path has the Up-Down structure, the problem can be simplified to finding the best path with this structure. This can be solved with a dynamic programming algorithm.

In the first step, two special paths are considered: a clockwise-numbered path and an anticlockwise-numbered path. Firstly, we consider the path that visits the nodes clockwise along the convex hull, namely the path $\langle 1, 2, 3, \ldots, n-1, n\rangle$. It can be seen that the path also follows the Up-Down structure, as all nodes visited from the depot to node i are in the upper set U^i, where $i = n$ and $U^n = \{2, 3, \ldots, n-2, n-1\}$. Secondly, we consider the path that visits nodes anticlockwise along the convex hull, namely the path $\langle 1, n, n-1, \ldots, 3, 2\rangle$. This path also follows the Up-Down structure, as all nodes visited from the depot to node i are in the lower set L^i, where $i = 2$ and $L^2 = \{n, n-1, \ldots, 4, 3\}$.

To calculate the cost of the two paths, denote V_i^1 as the length of the shortest sub-path from the depot to node i through $2, 3, \ldots, i-1$ (see Figure 4.7), where $i = 3, 4, \ldots, n$. When $i = n$, V_n^1 is the length of the clockwise-numbered path. Further, denote V_2^j as the length of the shortest sub-path from the depot to node 2 through $n, n-1, \ldots, j$ (see Figure 4.8), where $j = n, n-1, \ldots, 3$. When $j = 3$, V_2^3 is the length of the anticlockwise-numbered path. V_n^1 and V_2^3 can be calculated with the following recursions:

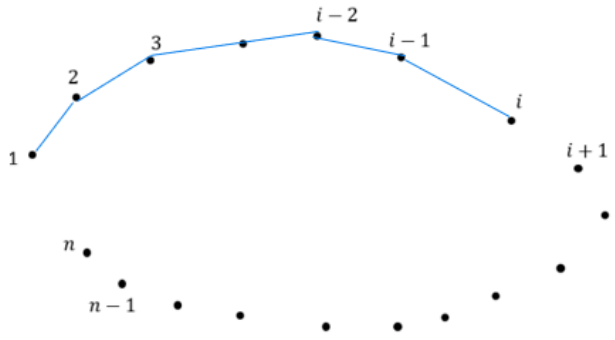


Figure 4.7: Demonstration of V_i^1, where V_n^1 is the length of the clockwise-numbered path

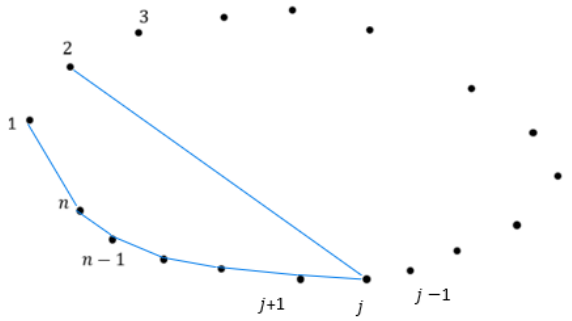


Figure 4.8: Demonstration of V_2^j, where V_2^3 is the length of the anticlockwise-numbered path

$$\begin{aligned} V_2^1 &= c_{1,2}, \\ V_i^1 &= V_{i-1}^1 + c_{i-1,i}, \text{ when } i = 3, 4, \ldots, n. \end{aligned} \tag{4.31}$$

$$\begin{aligned} V_2^n &= c_{1,n} + c_{n,2}, \\ V_2^j &= V_2^{j+1} - c_{j+1,2} + c_{j+1,j} + c_{j,2}, \text{ when } j = n-1, n-2, \ldots, 3. \end{aligned} \tag{4.32}$$

In the next step, we consider other paths with the special Up-Down structure, with part of the nodes in the upper set U^i and the remaining nodes in the lower set L^i, where $i \neq 2$ and $i \neq n$. Define V_i^j as the length of the shortest sub-path from the depot to node i through $2, 3, \ldots, i-1, n, n-1, \ldots, j$, where $i = 3, 4, \ldots, n-1; j = n, n-1, \ldots, i+1$.

In this case, there are two possible cases for the path from the depot to node i: the node preceding node i can be in U^i or L^i. The two cases are separately discussed below.

First case

If node i is reached directly from L^i, then the last edge in the shortest path is edge (j, i). This is demonstrated in Figure 4.9. Define $l_{a,b}$ as the length of the interval from a to b, where $a, b \in L^i$. In this case, V_i^j can be calculated with the following recursion:

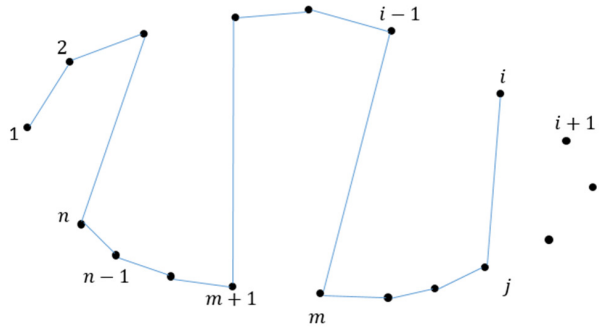


Figure 4.9: Demonstration of the first case of V_i^j: Visit i directly from L^i

$$
\begin{aligned}
&V_i^n = V_{i-1}^1 + c_{i-1,n} + c_{n,i}, \text{ when } i = 3, 4, \ldots, n-1. \\
&V_i^j = min \begin{cases} min(V_{i-1}^{m+1} + c_{i-1,m} + l_{m,j} + c_{j,i}), \ \forall\ m = j, j+1, \ldots, n-1; \\ V_{i-1}^1 + c_{i-1,n} + l_{n,j} + c_{j,i}, \end{cases} \\
&\text{when } i = 3, 4, \ldots, n-1; j = n-1, n-2, \ldots, i+1.
\end{aligned} \tag{4.33}
$$

The value of $l_{a,b}$ can be calculated with the following recursions:

$$
\begin{aligned}
&l_{j,j} = 0, \\
&l_{m,j} = c_{m,m-1} + l_{m-1,j}, \text{ when } j = 4, 5, \ldots, n-1; m = j+1, j+2, \ldots, n.
\end{aligned} \tag{4.34}
$$

Second case

If node i is reached directly from U^i (see Figure 4.10), then the last edge in the shortest path is edge $(i-1, i)$. Define $u_{c,d}$ as the length of the interval from c to d, where $c, d \in U^i$. In this case, V_i^j can be calculated with the following recursion:

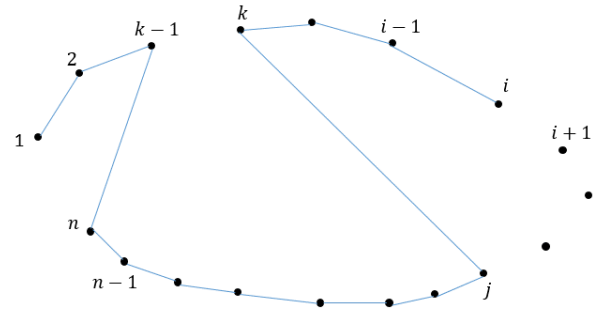


Figure 4.10: Demonstration of the second case of V_i^j: Visit i directly from U^i

$$V_i^j = min_{k=2,3,\ldots,i-1}(V_k^j + u_{k,i}) = V_{i-1}^j + c_{i-1,i} \text{ when } i = 3,\ldots,n-1; j = n,\ldots,i+1. \tag{4.35}$$

All possibilities of the paths with the Up-Down structure have now been considered. The optimal path should be the shortest path from the depot to node i through $2,3,\ldots,i-1,n,n-1,\ldots,i+1$, where $i = 3,4,\ldots,n-1$, or the clockwise-numbered path $\langle 1,2,3,\ldots,n-1,n\rangle$, or the anticlockwise-numbered path $\langle 1,n,n-1,\ldots,2\rangle$. All dynamic programming recursions are summarised below:

$$\begin{aligned}
&V_2^1 = c_{1,2}, V_2^n = c_{1,n} + c_{n,2},\\
&V_i^1 = V_{i-1}^1 + c_{i-1,i}, \text{ when } i = 3,4,\ldots,n.\\
&V_2^j = V_2^{j+1} - c_{j+1,2} + c_{j+1,j} + c_{j,2}, \text{ when } j = n-1, n-2,\ldots,3.\\
&V_i^j = min \begin{cases} V_{i-1}^1 + c_{i-1,n} + c_{n,i}; \\ V_{i-1}^j + c_{i-1,i}. \end{cases} \quad \text{when } i = 3,4,\ldots,n-1 \text{ and } j = n.\\
&V_i^j = min \begin{cases} min(V_{i-1}^{m+1} + c_{i-1,m} + l_{m,j} + c_{j,i}), \ \forall\ m = j, j+1,\ldots,n-1; \\ V_{i-1}^1 + c_{i-1,n} + l_{n,j} + c_{j,i}; \\ V_{i-1}^j + c_{i-1,i} \end{cases}\\
&\text{when } i = 3,4,\ldots,n-1 \text{ and } j = n-1, n-2,\ldots,i+1.\\
&Opt = min\{(min_{i=3,4,\ldots,n-1}(V_i^{i+1})), V_n^1, V_2^3\}.
\end{aligned} \tag{4.36}$$

The time complexity of calculating the length of the path $\langle 1,2,\ldots,n-1,n\rangle$ and the path $\langle 1,n,n-1,\ldots,2\rangle$ is $O(n)$, that of computing the values of $l_{a,b}$ is $O(n^2)$, and that of calculating V_i^{i+1} is $O(n^3)$. Therefore, the total time complexity is $O(n^3)$. We demonstrate the above recursions with a simple example. The coordinates of the nodes are provided in Table 4.1.

The length of the clockwise-numbered path $\langle 1,2,3,4,5,6\rangle$ and the length of the anticlockwise-numbered path $\langle 1,6,5,4,3,2\rangle$ are respectively calculated as V_6^1 and V_2^3 as follows:

Table 4.1: Simple example of the Path TSP on the convex hull: coordinates of nodes

point	1	2	3	4	5	6
x	0	0	4	7	6	4
y	1	3	6	4	1	0

$$V_2^1 = 2, V_3^1 = 7, V_4^1 = 10.61, V_5^1 = 13.77, V_6^1 = 16.01,$$
$$V_2^6 = 9.16, V_2^5 = 12.72, V_2^4 = 16.63, V_2^3 = 18.17.$$

Next, we calculate the lengths of the paths from the depot to node i through $2, 3, \ldots, i-1, n, n-1, \ldots, i+1$, where $i = 3, 4, \ldots, n-1$, and find the shortest one as follows. As described above, one first computes the value of $l_{a,b}$ and then the V_i^j values.

$L_{4,4} = L_{5,5} = 0, L_{5,4} = 3.16, L_{6,4} = 5.4, L_{6,5} = 2.24.$

$$V_3^6 = min \begin{cases} V_2^1 + c_{2,6} + c_{6,3} = 13 \\ V_2^6 + c_{2,3} = 14.16 \end{cases} = 13.$$

$$V_3^5 = min \begin{cases} V_2^6 + c_{2,5} + l_{5,5} + c_{5,3} = 20.87 \\ V_2^1 + c_{2,6} + l_{6,5} + c_{5,3} = 14.63 \\ V_2^5 + c_{2,3} = 17.72 \end{cases} = 14.63.$$

$$V_3^4 = min \begin{cases} min(V_2^5 + c_{2,4} + l_{4,4} + c_{4,3}, V_2^6 + c_{2,5} + l_{5,4} + c_{4,3}) = 22.25 \\ V_2^1 + c_{2,6} + l_{6,4} + c_{4,3} = 16.01 \\ V_2^4 + c_{2,3} = 21.63 \end{cases} = 16.01.$$

$$V_4^6 = min \begin{cases} V_3^1 + c_{3,6} + c_{6,4} = 18 \\ V_3^6 + c_{3,4} = 16.61 \end{cases} = 16.61.$$

$$V_4^5 = min \begin{cases} V_3^6 + c_{3,5} + l_{5,5} + c_{5,4} = 21.55 \\ V_3^1 + c_{3,6} + l_{6,5} + c_{5,4} = 18.4 \\ V_3^5 + c_{3,4} = 18.24 \end{cases} = 18.24.$$

$$V_5^6 = min \begin{cases} V_4^1 + c_{4,6} + c_{6,5} = 17.85 \\ V_4^6 + c_{4,5} = 19.77 \end{cases} = 17.85.$$

Therefore, the optimal value is 16.01, as shown below. The optimal TSP paths are $\langle 1, 2, 3, 4, 5, 6 \rangle$ and $\langle 1, 2, 6, 5, 4, 3 \rangle$, which are shown in Figure 4.11.

$$Opt = min(V_6^1, V_2^3, V_3^4, V_4^5, V_5^6) = min(16.01, 18.17, 16.01, 18.24, 17.85) = 16.01.$$

4.4.2 Up-Down Heuristic for CTSPP

We have found that the Path TSP on the convex hull is solvable with the Up-Down structure. This provides inspiration for the CTSPP. In the CTSPP, if the nodes are distributed along the convex hull or close to the convex hull, the best path among

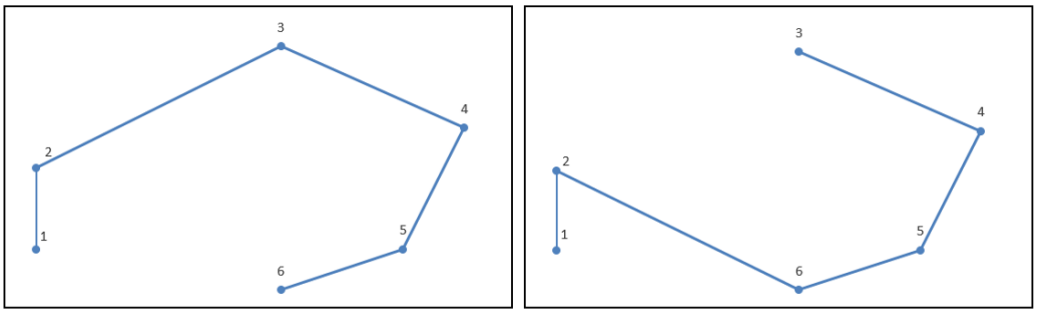


Figure 4.11: Numerical example of Path TSP on a convex hull: (a) optimal path 1; (b) optimal path 2

all paths with the Up-Down structure is expected to be a good solution. Therefore, we propose a heuristic called the Up-Down heuristic to find the best path with this special structure for the CTSPP.

The notations are similar to those in Section 4.4.1. The difference is that the term of 'length of a sub-path' in the Path TSP changes to 'cumulative cost of a sub-path' for the CTSPP. Specifically, denote V_i^1 as the smallest cumulative cost of the sub-path from the depot to node i through $2, 3, \ldots, i-1$, where $i = 3, 4, \ldots, n$. When $i = n$, V_n^1 is the cost of the clockwise-numbered path. Further, denote V_2^j as the smallest cumulative cost of the sub-path from the depot to node 2 through $n, n-1, \ldots, j$, where $j = n, n-1, \ldots, 3$. When $j = 3$, V_2^3 is the cost of the anticlockwise-numbered path. V_n^1 and V_2^3 can be calculated with the following recursions:

$$
\begin{aligned}
&V_2^1 = c_{1,2} \times (n-1), V_2^n = c_{1,n} \times (n-1) + c_{n,2} \times (n-2), \\
&V_i^1 = V_{i-1}^1 + c_{i-1,i} \times (n-i+1), \text{ when } i = 3, 4, \ldots, n. \\
&V_2^j = V_2^{j+1} + (c_{j+1,j} - c_{j+1,2}) \times (j-1) + c_{j,2} \times (j-2), \text{ when } j = n-1, n-2, \ldots, 3.
\end{aligned}
\tag{4.37}
$$

Note that the recursions are similar to those in Section 4.4.1. The difference is that we incorporate the coefficients of edges in the calculation of the cumulative costs. For example, V_i^1 is obtained from V_{i-1}^1 and the edge $(i-1, i)$. Because $i-1$ is the $(i-1)^{th}$ node, the coefficient should be $n-(i-1)$. However, for the Path TSP, all the coefficients are 1 in the recursions.

For other paths with the Up-Down structure, denote V_i^j as the smallest cumulative cost of the sub-path from the depot to node i through $2, 3, \ldots, i-1, n, n-1, \ldots, j$, where all nodes are visited in the same order as in U^i and L^i, and $i = 3, 4, \ldots, n-1; j = n, n-1, \ldots, i+1$. Figure 4.9 shows one case: node i is reached directly from L^i. In this case, $l_{m,j,i}$ is defined as the cumulative cost of the interval from m to j, where node j is linked to node i, and $m, j \in L^i$. Figure 4.10 demonstrates the other case: node i is reached directly from U^i. In this case, define $u_{k,i,j}$ as the cumulative cost of the interval from k to i, where node k is linked to node j, and $k \in U^i$. The recursions for calculating V_i^j are given below:

$$
\begin{aligned}
V_i^n &= min \begin{cases} V_{i-1}^1 + c_{i-1,n} \times (n-i+1) + c_{n,i} \times (n-i); \\ min(V_k^n + u_{k,i,n}),\ \forall\ k = i-1, i-2, \ldots, 2; \end{cases} \quad \text{when } i = 3, 4, \ldots, n-1. \\
V_i^j &= min \begin{cases} min(V_{i-1}^{m+1} + c_{i-1,m} \times (m-i+1) + l_{m,j,i} + c_{j,i} \times (j-i)),\ \forall\ m = j, \ldots, n-1; \\ V_{i-1}^1 + c_{i-1,n} \times (n-i+1) + l_{n,j,i} + c_{j,i} \times (j-i); \\ min(V_k^j + u_{k,i,j}),\ \forall\ k = i-1, i-2, \ldots, 2; \end{cases} \\
&\text{when } i = 3, 4, \ldots, n-1 \text{ and } j = n-1, n-2, \ldots, i+1.
\end{aligned}
\tag{4.38}
$$

Again, the values of $l_{m,j,i}$ and $u_{k,i,j}$ need to be calculated to determine the costs of the intervals. The difference from the Path TSP is that the cost here is the cumulative value. The following recursions can be used to obtain the required values:

$$
\begin{aligned}
&l_{j,j,i} = 0, \\
&l_{m,j,i} = c_{m,m-1} \times (m-i) + l_{m-1,j,i}, \\
&\text{when } i = 3, 4, \ldots, n-1, j = n-1, n-2 \ldots, i+1, m = j+1, j+2, \ldots, n. \\
&u_{i,i,j} = 0, \\
&u_{k,i,j} = c_{i-1,i} \times (j-i) + u_{k,i-1,j}, \\
&\text{when } i = 3, 4, \ldots, n-1, j = n, n-1 \ldots, i+1, k = i-1, i-2, \ldots, 2.
\end{aligned}
\tag{4.39}
$$

We have now considered all possibilities of the paths with the Up-Down structure for the CTSPP: paths from the depot to node i through $2, 3, \ldots, i-$

$1, n, n-1, \ldots, i+1$, where $i = 3, 4, \ldots, n-1$, the clockwise-numbered path $\langle 1, 2, 3, \ldots, n-1, n \rangle$ and the anticlockwise-numbered path $\langle 1, n, n-1, \ldots, 2 \rangle$. Therefore, the cost of the best path with the Up-Down structure is as follows, where the time complexity is also $O(n^3)$.

$$Opt = min\{(min_{i=3,4,\ldots,n-1}(V_i^{i+1})), V_n^1, V_2^3\} \tag{4.40}$$

4.4.3 Two-Ray Heuristic for CTSPP

Next, we consider a more special case, a two-ray case. In this special case, the n nodes are distributed along two rays: the upper ray $\langle 2, 3, \ldots, UE \rangle$ and the lower ray $\langle n, n-1, \ldots, LE \rangle$, with the depot, node 1, as the intersection. This structure is demonstrated in Figure 4.12. We denote UE and LE as the right-end point in the upper ray and lower ray respectively.

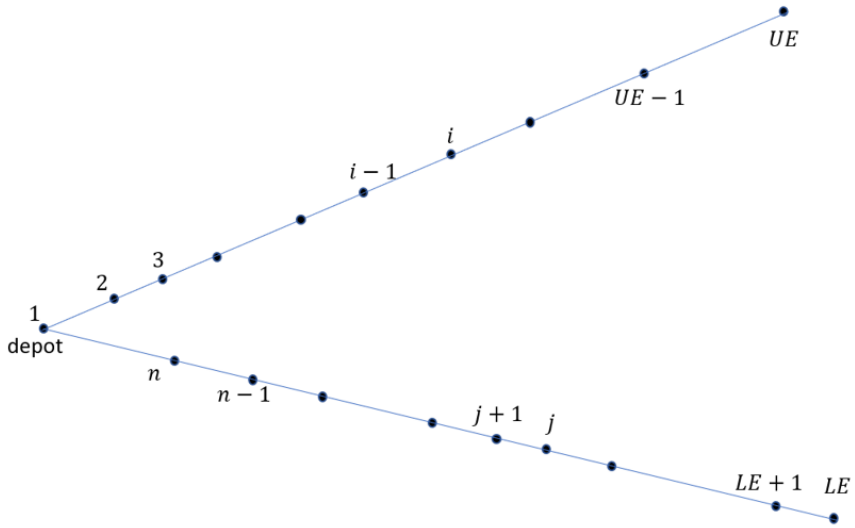


Figure 4.12: Illustration of the node distribution along two rays

In this section, the Two-Ray heuristic is proposed for the CTSPP. This heuristic combines the Up-Down structure and the Line structure in the solvable Line-CTSPP. We propose the Two-Ray heuristic for two reasons. Firstly, the two-ray case can be seen as a special case of the convex hull when $LE = UE + 1$. Considering that the Up-Down heuristic performs well on the convex-hull cases (as shown in the experiment in Section 4.5), the path with the Up-Down structure is expected to deliver a good performance on the two-ray case. Secondly, when the nodes on the upper ray and lower ray are partitioned into intervals, all the intervals

are straight lines. Each interval can be regarded as a Part CTSPP Line problem, which was solved in Section 4.2.3.

We first explain the Up-Down structure in this heuristic. Consecutive nodes on the upper ray can be used to construct upper intervals and, similarly, lower intervals are composed of consecutive nodes on the lower ray. Assume that all nodes are partitioned into intervals in order and that there is no overlap of nodes. Denote the lists of upper and lower intervals as follows. For example, assuming $UE = 6, n = 10$, then one set of reasonable intervals can be $UI_1 = \langle 2, 3\rangle, UI_2 = \langle 4, 5, 6\rangle, LI_1 = \langle 10\rangle, LI_2 = \langle 9, 8, 7\rangle$.

$$
\begin{aligned}
&UI_1 = \langle k_1, k_1 + 1, \ldots, i_1\rangle,\\
&UI_2 = \langle k_2, k_2 + 1, \ldots, i_2\rangle,\\
&\ldots,\\
&UI_t = \langle k_t, k_t + 1, \ldots, i_t\rangle,\\
&\text{where } k_1 = 2, k_2 = i_1 + 1, \ldots, i_t = UE;\\
&LI_1 = \langle m_1, m_1 - 1, \ldots, j_1\rangle,\\
&LI_2 = \langle m_2, m_2 - 1, \ldots, j_2\rangle,\\
&\ldots,\\
&LI_r = \langle m_r, m_r - 1, \ldots, j_r\rangle,\\
&\text{where } m_1 = n, m_2 = j_1 - 1, \ldots, j_r = LE.
\end{aligned}
$$

The Up-Down structure in this heuristic means that when visiting nodes, upper intervals alternate with lower intervals, and all intervals are visited in order, i.e., nodes in UI_{a+1} cannot be visited earlier than nodes in UI_a, and nodes in LI_{b+1} cannot be visited earlier than nodes in LI_b. As demonstrated in the example above, the path $\langle 1, UI_1, LI_1, UI_2, LI_2\rangle$ satisfies the Up-Down structure, while the path $\langle 1, UI_1, LI_2, UI_2, LI_1\rangle$ violates the Up-Down structure.

We then demonstrate the Line structure in the Two-Ray heuristic. The basic idea is that each interval is processed similarly to in the Part CTSPP Line in Section 4.2.3. When an interval is visited, assume that the first visited node (called the depot of an interval) is d, the exit node is e, and d is the p^{th} node in the complete path. Then we can find the optimal sub-path that can minimise the cost of all nodes on this interval.

Having explained the Up-Down structure and the Line structure, the Two-

Ray heuristic can be summarised as follows: consider all paths where the lower and upper intervals are visited in order and all intervals are processed similarly to in Line-CTSPP, then choose the best path with the minimum cost.

To construct dynamic programming recursions, we make the following definitions. The upper interval is denoted as $UI(k, i, d, e, p)$. There are five parameters: left-end point k, right-end point i, depot d, exit point e and depot position p. In other words, any path through this upper interval starts from depot d, visits the set of nodes $k, k+1, \ldots, i-1, i$ and ends at the exit point e, and d is the p^{th} node in the complete path. One upper interval can be seen in Figure 4.13. The optimal path through this upper interval is denoted as $UP(k, i, d, e, p)$ and the optimal cost (i.e., the total contribution to the complete path) is denoted as $CU(k, i, d, e, p)$.

Similarly, the lower interval is denoted as $LI(m, j, d', e', p')$ with left-end point m, right-end point j, depot d', exit point e' and depot position p'. The paths through this lower interval start from d', visit the set of nodes $m, m-1, \ldots, j+1, j$ and end at e'. One lower interval can be seen in Figure 4.14. The optimal path through this lower interval and the optimal cost are denoted as $LP(m, j, d', e', p')$ and $CL(m, j, d', e', p')$ respectively.

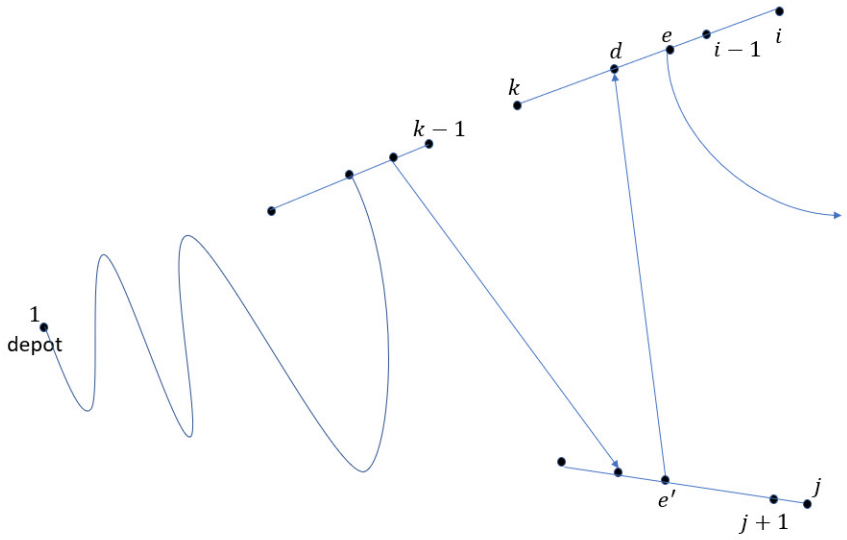


Figure 4.13: Illustration of $U(i, j, e)$, the cost of the lowest-cost sub-path through $1, 2, \ldots, i-1, i, j, j+1, \ldots, n$, ending at e in an upper interval

In addition, define $U(i, j, e)$ as the cost of the lowest-cost sub-path through $1, 2, \ldots, i-1, i, j, j+1, \ldots, n$, ending at the exit point e in the upper ray. This sub-path is illustrated in Figure 4.13. Similarly, $L(i, j, e')$ is defined as the cost of the lowest-cost sub-path through $1, 2, \ldots, i-1, i, j, j+1, \ldots, n$ with the exit point

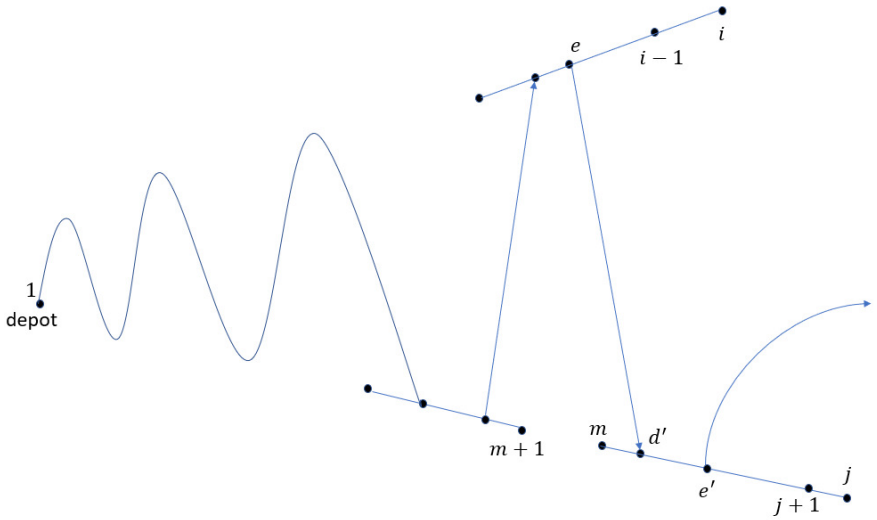


Figure 4.14: Illustration of $L(i, j, e')$, the cost of the lowest-cost sub-path through $1, 2, \ldots, i-1, i, j, j+1, \ldots, n$, ending at e' in a lower interval

e' in the lower ray. This sub-path can be seen in Figure 4.14. Then one can write the following dynamic programming recursions:

When $i = 2, 3, \ldots, UE,\ j = n, n-1, \ldots, LE,\ e = 2, 3, \ldots, i$:
$U(i, j, e) = min(L(k-1, j, e') + c(e', d) \times (j - k) + CU(k, i, d, e, p))$
for all $2 \leq k \leq e, k \leq d \leq i, j \leq e' \leq n$;

When $i = 2, 3, \ldots, UE,\ j = n, n-1, \ldots, LE,\ e' = n, n-1, \ldots, j$:

$$L(i, j, e') = \begin{cases} min(U(i, m+1, e) + c(e, d') \times (m - i) + CL(m, j, d', e', p')) \\ \text{for all } e' \leq m \leq n-1, j \leq d' \leq m, 2 \leq e \leq i \\ \\ min(U(i, 1, e) + c(e, d') \times (n - i) + CL(n, j, d', e', p')) \\ \text{for all } j \leq d' \leq n, 2 \leq e \leq i \end{cases} \tag{4.41}$$

In the first recursion, the optimal path ending on the upper interval first visits the optimal path ending on the last lower interval, with the exit point e' on the last lower interval linked to depot d on the upper interval, and then visits

the upper interval with the optimal path $UP(k, i, d, e, p)$. Here, $L(k-1, j, e')$ is the cost of the lowest-cost path through $1, 2, \ldots, k-1, j, j+1, \ldots, n$ with the exit point e'. Therefore, e' is the $(n+k-j)^{th}$ point; thus, the coefficient of edge (e', d) is $n-(n+k-j) = j-k$. Also, depot d is the $(n+k-j+1)^{th}$ point; thus, $p = n+k-j+1$.

Similarly, $L(i, j, e')$ can be written as the sum of $U(i, m+1, e)$ (or $U(i, 1, e)$ when $m = n$), the cost of the edge (e, d') and the cost of the lower path $LP(m, j, d', e', p')$. Here, $U(i, m+1, e)$ is the cost of the lowest-cost path through $1, 2, \ldots, i, m+1, m+2, \ldots, n$ with the exit point e. Therefore, the positions of e and d' are $n+i-m$ and $n+i-m+1$ respectively; thus, the coefficient of edge (e, d') is $m-i$ and $p' = n+i-m+1$.

Then we consider initial conditions where the sub-path visits nodes only on the upper ray or only on the lower ray. Note that node 1 is on both the upper ray and the lower ray. The following recursions describe the initial conditions:

$$
\begin{aligned}
&U(i, 1, e) = CU(1, i, 1, e, 1) \text{ for all } 1 \le i \le UE, 1 \le e \le i;\\
&L(1, n, n) = c(1, n) \times (n-1);\\
&L(1, j, e') = c(1, n) \times (n-1) + CL(n, j, n, e', 2) \text{ for all } LE \le j \le n-1, j \le e' \le n-1;\\
&L(1, j, n) = c(1, n-1) \times (n-1) + CL(n, j, n-1, n, 2) \text{ for all } LE \le j \le n-1.
\end{aligned}
\tag{4.42}
$$

Then the optimal value can be written as follows:

$$
\begin{aligned}
&Opt = min(U(UE, LE, e), L(UE, LE, e'))\\
&\text{for all } 2 \le e \le UE, LE \le e' \le n
\end{aligned}
\tag{4.43}
$$

Now, we demonstrate the recursions using a simple example with $n = 5, UE = 3$. The coordinates of the nodes are provided in Table 4.2, and the calculations are given below.

Table 4.2: Simple example of the CTSPP on a two-ray case: coordinates of nodes

point	1	2	3	4	5
x	0	2	6	4	2
y	0	1	3	0	0

$U(1,1,1) = 0, U(2,1,1) = \infty, U(2,1,2) = c(1,2) \times 4 = 8.94, U(3,1,1) = \infty,$

$U(3,1,2) = c(1,3) \times 4 + c(3,2) \times 3 = 40.25, U(3,1,3) = c(1,2) \times 4 + c(2,3) \times 3 = 22.36.$

$L(1,5,5) = c(1,5) \times 4 = 8,$

$L(1,4,5) = c(1,4) \times 4 + c(4,5) \times 3 = 22,$

$L(1,4,4) = c(1,5) \times 4 + c(5,4) \times 3 = 14.$

$U(2,5,2) = L(1,5,5) + c(5,2) \times 3 + CU(2,2,2,2,3) = 11.$

$L(2,5,5) = U(2,1,1) + c(2,5) \times 3 + CL(5,5,5,5,3) = 11.94.$

$$U(2,4,2) = min \begin{cases} L(1,4,4) + c(4,2) \times 2 + CU(2,2,2,2,4) = 18.47 \\ L(1,4,5) + c(5,2) \times 2 + CU(2,2,2,2,4) = 24 \end{cases} = 18.47.$$

$$L(2,4,5) = min \begin{cases} U(2,1,2) + c(2,5) \times 3 + CL(5,4,5,5,3) = \infty \\ U(2,1,2) + c(2,4) \times 3 + CL(5,4,4,5,3) = 19.66 \end{cases} = 19.66.$$

$$L(2,4,4) = min \begin{cases} U(2,5,2) + c(2,4) \times 2 + CL(4,4,4,4,4) = 15.47 \\ U(2,1,2) + c(2,5) \times 3 + CL(5,4,5,4,3) = 15.94 \\ U(2,1,2) + c(2,4) \times 3 + CL(5,4,4,4,3) = \infty \end{cases} = 15.47.$$

$$U(3,5,2) = min \begin{cases} L(1,5,5) + c(5,2) \times 3 + CU(2,3,2,2,3) = \infty \\ L(1,5,5) + c(5,3) \times 3 + CU(2,3,3,2,3) = 31.94 \end{cases} = 31.94.$$

$$U(3,5,3) = min \begin{cases} L(2,5,5) + c(5,3) \times 2 + CU(3,3,3,3,4) = 21.94 \\ L(1,5,5) + c(5,2) \times 3 + CU(2,3,2,3,3) = 19.94 \\ L(1,5,5) + c(5,3) \times 3 + CU(2,3,3,3,3) = \infty \end{cases} = 19.94.$$

$$L(3,5,5) = min \begin{cases} U(3,1,3) + c(3,5) \times 2 + CL(5,5,5,5,4) = 32.36 \\ U(3,1,2) + c(2,5) \times 2 + CL(5,5,5,5,4) = 42.25 \end{cases} = 32.36.$$

$$U(3,4,2) = min \begin{cases} L(1,4,4) + c(4,2) \times 2 + CU(2,3,2,2,4) = \infty \\ L(1,4,5) + c(5,2) \times 2 + CU(2,3,2,2,4) = \infty \\ L(1,4,4) + c(4,3) \times 2 + CU(2,3,3,2,4) = 25.68 \\ L(1,4,5) + c(5,3) \times 2 + CU(2,3,3,2,4) = 36.47 \end{cases} = 25.68.$$

$$U(3,4,3) = min \begin{cases} L(2,4,4) + c(4,3) \times 1 + CU(3,3,3,3,5) = 19.08 \\ L(2,4,5) + c(5,3) \times 1 + CU(3,3,3,3,5) = 24.65 \\ L(1,4,4) + c(4,2) \times 2 + CU(2,3,2,3,4) = 22.94 \\ L(1,4,5) + c(5,2) \times 2 + CU(2,3,2,3,4) = 28.47 \\ L(1,4,4) + c(4,3) \times 2 + CU(2,3,3,3,4) = \infty \\ L(1,4,5) + c(5,3) \times 2 + CU(2,3,3,3,4) = \infty \end{cases} = 19.08.$$

$$L(3,4,5) = min \begin{cases} U(3,1,3) + c(3,5) \times 2 + CL(5,4,5,5,4) = \infty \\ U(3,1,2) + c(2,5) \times 2 + CL(5,4,5,5,4) = \infty \\ U(3,1,3) + c(3,4) \times 2 + CL(5,4,4,5,4) = 31.57 \\ U(3,1,2) + c(2,4) \times 2 + CL(5,4,4,5,4) = 46.72 \end{cases} = 31.57.$$

$$L(3,4,4) = min \begin{cases} U(3,5,3) + c(3,4) \times 1 + CL(4,4,4,4,5) = 23.55 \\ U(3,5,2) + c(2,4) \times 1 + CL(4,4,4,4,5) = 34.18 \\ U(3,1,3) + c(3,5) \times 2 + CL(5,4,5,4,4) = 34.36 \\ U(3,1,2) + c(2,5) \times 2 + CL(5,4,5,4,4) = 44.24 \\ U(3,1,3) + c(3,4) \times 2 + CL(5,4,4,4,4) = \infty \\ U(3,1,2) + c(2,4) \times 2 + CL(5,4,4,4,4) = \infty \end{cases} = 23.55.$$

Therefore, the optimal value is 19.08, as shown below.

$$Opt = min(U(3,4,2), U(3,4,3), L(3,4,5), L(3,4,4)) = 19.08.$$

Assuming that the optimal cumulative costs of all intervals are already known, then the time complexity is $O(n^6)$, because $O(n^3)$ values need to be calculated and calculating each value requires $O(n^3)$ time. $O(n^2)$ time is required to calculate the optimal costs for the upper and lower intervals. Therefore, the total time complexity is $O(n^6)$. We next show how to calculate the optimal costs for the intervals.

Given an upper interval $UI(a, b, d, e, p)$ with left-end point a, right-end point b, depot d, exit point e and depot position p, the nodes are first split into two parts: left-side nodes and right-side nodes, as shown in Figure 4.15. We number the nodes

as in Figure 4.5 for Part CTSPP Line with a fixed exit point in Section 4.2.3, where a is y_m and b is x_k. Then the optimal costs for this interval can be calculated with the recursions as in Section 4.2.3.

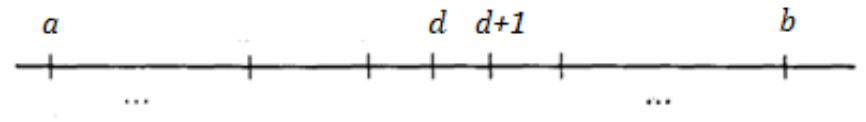


Figure 4.15: Illustration of node distribution of the upper interval $UI(a, b, d, e, p)$

We then apply the parameters of $CU(k, i, d, e, p)$ to the recursions in Section 4.2.3 to obtain the optimal costs for all upper intervals. The calculations for the lower intervals are similar. We have now discussed all recursions in the Two-Ray heuristic.

4.5 Computational Experiments

To test the performance of proposed heuristics, they are coded in C++ and executed on Intel Core i5-10600 3.30 GHz processor with 32.0 GB RAM.

4.5.1 Experiment Design

We aim to show the power of the proposed heuristics on specially structured cases and on cases with structures in a sense 'close' to these structures. Here, 'close' means although the nodes do not follow the special structures exactly, the nodes are scattered closely around the special distributions or only a small proportion of nodes violate the special distributions. This will be illustrated in the two-line cases and the close-to-convex-hull cases respectively.

The experiments all involve self-generated instances comprising two-line instances, convex-hull instances, close-to-convex-hull cases and two-ray instances. The generation process for each instance set is described below.

Two-line instances As the Line heuristic is based on the solvable line case, it can be expected that the proposed heuristic will perform well on close-to-line cases. Here, 'close' means that the nodes are scattered closely around the special distribution: the straight line. We consider a case where the nodes are distributed along two parallel lines with a small distance between the lines. This makes the node area a narrow strip, and thus close to a line. This type of case has practical applications, such as delivery to customers located along two sides of one street. We

generate 30 two-line instances for five instance sets with sizes 20, 50, 100, 200 and 500, which are used in the first experiment. The generation is performed as follows: The X-coordinate is randomly chosen from the range $[0, x]$ based on the size: x is set as 100, 200, 500, 1000 and 2000 for sizes 20, 50, 100, 200 and 500 respectively. To ensure that all nodes are distributed along two lines, the Y-coordinate is chosen randomly from only two values: 0 and y. In the experiment, y is set as 3 to make the node area a narrow strip. Distances are Euclidean distances rounded to the nearest integer.

Convex-hull instances Twenty random convex-hull instances are generated for each of the problem sizes of 20, 50, 100, 200 and 500 for use in the second experiment. The generation is based on Valtr's proof in [143] to produce convex polygons as follows. First, two lists of random X and Y coordinates are generated, then the coordinates are sorted. After isolating the extreme points, the interior points are divided into two chains randomly. Then the vector components are extracted, the X and Y components are paired randomly and the paired components are combined into vectors. The vectors are sorted by angle and linked end-to-end to form a polygon. Finally, the polygon is moved so that one vertex lies on the origin. Note that the ranges of the coordinates vary with the size and are 100, 500, 500, 1000 and 5000 for sizes 20, 50, 100, 200 and 500 respectively. Distances are also Euclidean distances rounded to the nearest integer.

Close-to-convex-hull instances For each of the problem sizes 20, 50, 100, 200 and 500, we generate 20 random close-to-convex-hull instances, which are based on the convex-hull instances above. Here, 'close' means that only a small proportion of nodes violate the original special distribution. Given a convex-hull instance, the idea is to select a small number of nodes randomly and modify the coordinates based on their original positions. To better illustrate the meaning of 'close', a comparison of one convex-hull instance with n=20 and its corresponding close-to-convex-hull instance is given in Figure 4.16.

Specifically, we first determine the turning points where the increasing or decreasing trend of coordinates changes in the convex hull. There are usually four turning points; thus, we can partition the nodes into four parts. At the turning points, the trend of the X- or Y-coordinate changes. If we choose nodes from the previous turning point to a turning point where the trend of the X-coordinate changes, then the Y-coordinate of the chosen nodes remains unchanged, whereas the X-coordinate increases by a; if we choose nodes from the previous turning point to a turning point where the trend of the Y-coordinate changes, then the X-coordinate of

the chosen nodes remains unchanged, whereas the Y-coordinate increases by a. For example, in the instance in Figure 4.16, from the depot $(0, 0)$, the X-coordinate first increases and the Y-coordinate increases (or remains constant), and the first turning point is found where the Y-coordinate starts to decrease. If nodes in the first part are chosen, then the X-coordinate remains unchanged, whereas the Y-coordinate increases by a. The verification of the other turning points and the modification of the coordinates are based on the same rules. To ensure that the instances are close-to-convex-hull instances, we only modify a small proportion of nodes: 3, 5, 5, 10 and 10 nodes for size-20, -50, -100, -200 and -500 instances respectively. The value of a is a random number in the range $[-b, b]$, where b is 10, 20, 20, 40 and 100 for size-20, -50, -100, -200 and -500 instances respectively. Distances are also Euclidean distances rounded to the nearest integer. These instances are used in the second experiment.

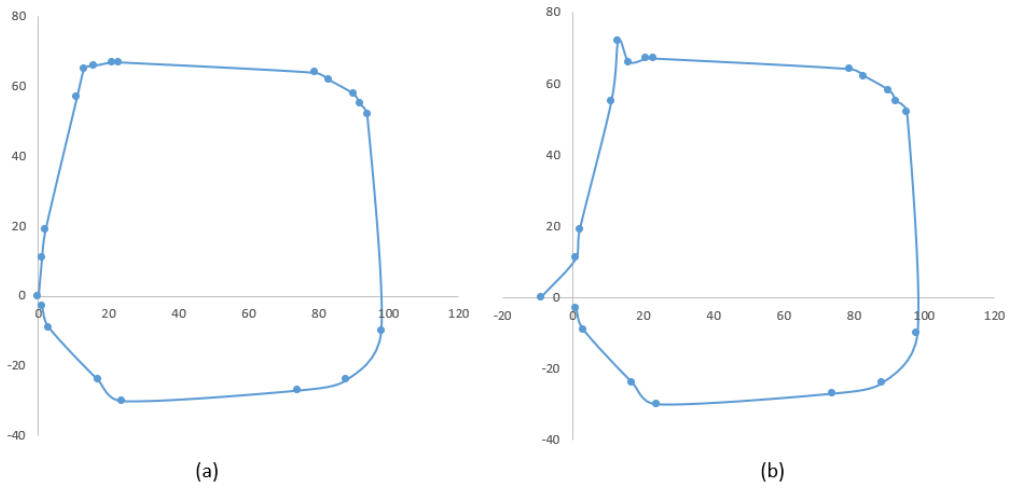


Figure 4.16: Comparison of one convex-hull instance and the corresponding close-to-convex-hull instance with n=20: (a) convex-hull instance; (b) close-to-convex-hull instance

Two-ray instances We generate size-24, -30 and -50 instances. Each size has 60 instances. For each size, three different slopes s (i.e., the ratio of the Y-coordinate to the X-coordinate) are used, 0.5, 1 and 2, where each slope contains 20 instances. Given a size n and a slope s, we first determine the number of nodes u on the upper ray. Two groups are generated. In the first group, called the Random group, u is randomly chosen from the range $[1, n]$; in the second group, called the Even group, $u = n/2$, that is, the number of nodes is the same on the upper and lower rays. We denote $(n, s, Random)$ and $(n, s, Even)$ as a Random instance set

and an Even instance set with size n, slope s and nodes from the Random group and Even group respectively. Each instance set contains 10 instances. Then the coordinates of each node are determined in the following way. The X-coordinate is randomly chosen from the range $[0, x]$, where x is set as 100, 100 and 500 for sizes 24, 30 and 50 respectively, and the Y-coordinate is determined on the basis of the slope. Then the nodes are numbered along the two rays as shown in Figure 4.12. The instances are used in the third experiment.

First Experiment

The purpose of the first experiment is to test the Line heuristic on two-line instances. As discussed in Chapter 3, the classical 2-opt obtains the solutions with the highest quality, and GRASP is commonly used for tour construction; thus the Line heuristic is compared with GRASP-2-opt.

For each instance using each algorithm, the objective value v and the running time t are recorded. For size-50, -100, -200 and -500 instances, the best solution obtained from the Line heuristic and GRASP-2-opt is chosen as the comparison base v_{best} for each instance, and the percentage gap $g\% = \frac{v - v_{best}}{v_{best}} \times 100\%$ is calculated as the solution quality, where clearly the smaller the percentage gap, the better the solution quality. Note that for instances of size 20, the exact optimal solution v_{opt} can be obtained; thus, the percentage gap is calculated as $g^*\% = \frac{v - v_{opt}}{v_{opt}} \times 100\%$. For each instance set, the average values of t, $g\%$ and $g^*\%$, respectively denoted as T, $G\%$ and $G^*\%$, are calculated.

We compare both the solution quality and the running time and test whether one algorithm is dominated by another. In this experiment, the single-start is adopted for GRASP-2-opt.

Second Experiment

This experiment is conducted to present the performance of the Up-Down heuristic on specially structured cases. Again, we compare the proposed heuristic with GRASP-2-opt.

Firstly, the Up-Down heuristic is compared with the classical GRASP-2-opt on convex-hull cases. Both single-start and multi-start are adopted for GRASP-2-opt, which are recorded as GRASP-2-opt(S) and GRASP-2-opt(M) respectively in the experimental results. We compare the Up-Down heuristic with GRASP-2-opt(S) and conduct the domination analysis. Then, the multi-start is adopted for GRASP-2-opt to ensure the same running time as that for the Up-Down heuristic. By controlling the running time, the solution quality becomes the key to evaluating the performance of algorithms. The measurement of the solution quality is similar

to that in the first experiment: the comparison base for size-50, -100, -200 and -500 instances is the best solution obtained from the two heuristics, while for instances of size 20, the optimal solution is used as the comparison base. Secondly, we perform experiments on close-to-convex-hull cases, where the experimental process is the same as that for convex-hull cases.

Third Experiment

The third experiment is conducted on two-ray instances to demonstrate the potential of the Two-Ray heuristic. Different from the other proposed heuristics in this chapter, our objective is not to outperform the classical heuristics in both solution quality and running time. Instead, the aim is to show that the Two-Ray heuristic can obtain high-quality solutions when the nodes have a special distribution along two rays. As discussed in Chapter 3, RIH-$S(3,1)$ has the highest performance in terms of solution quality on general cases. As shown in this section, the Two-Ray heuristic obtains a better solution quality than RIH-$S(3,1)$ on this specially structured case. We investigate how many instances can be solved to optimality with the Two-Ray heuristic for a size of 24. We also reveal the number of instances improved by the Two-Ray heuristic compared with RIH-$S(3,1)$ when the size is 30 and 50.

4.5.2 Experimental Results and Implications

First Experiment

The experimental results on two-line cases are presented in Table 4.3. The results show the average running time T and solution quality $G\%$ (or $G^*\%$) for each instance group. As discussed in Chapter 3, one heuristic is dominated by another if the other heuristic generates the same solutions with less time, or generates better solutions with the same time, or is superior in both solution quality and running time. From Table 4.3, the running time of the Line heuristic is less than that of GRASP-2-opt, and the percentage gap of the Line heuristic is smaller than that of GRASP-2-opt for all five instance groups, which suggests that the Line heuristic can obtain better solutions than GRASP-2-opt in less time for all the experimental instance sets. The large difference in the percentage gap $(20 - 35\%)$ between the two heuristics suggests that the Line heuristic is promising in special cases where the node distribution resembles a line.

Second Experiment

The experimental results on convex-hull cases and close-to-convex-hull cases

Table 4.3: Comparison of results of Line heuristic and GRASP-2-opt with single-start on two-line instances

		Line heuristic	GRASP-2-opt
$n = 20$	$T(s)$	8.30×10^{-6}	2.26×10^{-5}
	$G^*\%$	0.49	36.83
$n = 50$	$T(s)$	2.65×10^{-5}	1.72×10^{-4}
	$G\%$	0	29.28
$n = 100$	$T(s)$	6.26×10^{-5}	1.12×10^{-3}
	$G\%$	0	24.61
$n = 200$	$T(s)$	1.61×10^{-4}	8.03×10^{-3}
	$G\%$	0	23.31
$n = 500$	$T(s)$	1.31×10^{-3}	1.37×10^{-1}
	$G\%$	0	20.10

are presented in Tables 4.4 and 4.5 respectively.

In general, the results found in the two tables are similar. There is no evident domination relationship between the Up-Down heuristic and GRASP-2-opt(S): the proposed heuristic obtains better solutions but with a longer running time. Nevertheless, the Up-Down heuristic still demonstrates its potential on the specially structured cases as the solution quality is much better than that of GRASP-2-opt(S) with a slightly longer running time. To remove the effects of the running time, the multi-start is next adopted for GRASP-2-opt to control the time. The results in the third column and the last column in the two tables suggest that, with the running time controlled, the average percentage gap of the Up-Down heuristic is smaller than that of GRASP-2-opt(M) for all five instance groups. The third column, with $G\% = 0$ throughout, also indicates that the Up-Down heuristic obtains better results for all single instances when the size is 50, 100, 200 and 500. These results all demonstrate the superiority of the Up-Down heuristic to GRASP-2-opt on the special cases.

In addition, when testing on convex-hull cases, the percentage gap $G^*\% = 0$ shows that the Up-Down heuristic can solve the CTSPP to optimality when the size is 20. For the close-to-convex-hull cases, although not all solutions are optimal when the size is 20, the small $G^*\%$ still indicates the good performance of the Up-Down heuristic. This suggests that the Up-Down heuristic performs similarly

in convex-hull cases and close-to-convex-hull cases.

Table 4.4: Comparison of results of Up-Down heuristic and GRASP-2-opt (with/without controlling running time) on convex-hull instances

		Up-Down	GRASP-2-opt(S)	GRASP-2opt(M)
$n = 20$	$T(\times 10^{-5}s)$	2.33	2.30	3.11
	$G^*\%$	0	38.88	28.45
$n = 50$	$T(\times 10^{-4}s)$	2.79	1.60	3.49
	$G\%$	0	35.82	19.77
$n = 100$	$T(\times 10^{-3}s)$	2.59	1.05	3.02
	$G\%$	0	28.32	17.28
$n = 200$	$T(\times 10^{-3}s)$	24.18	7.78	27.47
	$G\%$	0	27.92	17.56
$n = 500$	$T(\times 10^{-1}s)$	6.62	1.31	7.18
	$G\%$	0	30.72	18.58

Table 4.5: Comparison of results of Up-Down heuristic and GRASP-2-opt (with/without controlling running time) on close-to-convex-hull instances

		Up-Down	GRASP-2-opt(S)	GRASP-2opt(M)
$n = 20$	$T(\times 10^{-5}s)$	2.52	2.35	3.20
	$G^*\%$	0.04	36.94	26.14
$n = 50$	$T(\times 10^{-4}s)$	3.45	1.87	3.90
	$G\%$	0	28.95	17.45
$n = 100$	$T(\times 10^{-3}s)$	3.05	1.16	3.44
	$G\%$	0	23.67	15.29
$n = 200$	$T(\times 10^{-3}s)$	24.41	8.14	28.27
	$G\%$	0	23.41	15.59
$n = 500$	$T(\times 10^{-1}s)$	6.82	1.34	7.30
	$G\%$	0	25.12	17.44

Third Experiment

The experimental results on two-ray cases are presented in Tables 4.6 and 4.7. When testing on $n = 24$ instances, the Two-Ray heuristic is expected

to solve the CTSPP on two-ray cases to optimality as the proposed heuristic considers the structural properties of the cases. Table 4.6 provides the number of instances solved to optimality for each group, which suggests that although the heuristic performs well in many cases, there are still 'difficult' cases where the optimal solution is not found. Figure 4.17 and 4.18 compare the exact optimal route with the Two-Ray heuristic route on two 'difficult' two-ray cases. For these 'difficult' cases, the upper and lower intervals are not visited in order in the optimal routes, which violates the Up-Down property in the Two-Ray heuristic. They indicate that it is possible for a more sophisticated heuristic to solve the 'difficult' cases if it considers paths that go one way from the depot, the intersection of two rays, with the Up-Down structure and then change direction and return to the intersection with the Up-Down structure. However, we found another more 'difficult' instance shown in Figure 4.19. This instance indicates that the path structure in the more sophisticated heuristic is also violated in the optimal path: the path goes one way from the depot with the Up-Down structure, then changes direction and returns to the depot with the Up-Down structure, and then changes direction for a third time. The coordinates of these 'difficult' cases can be seen in Appendix C.

The Two-Ray heuristic and the potential sophisticated heuristic consider the important structural properties of the cases; thus, the 'difficult' cases we found now allow us to formulate the conjecture that the CTSPP is NP-hard for two-ray cases. Although this conjecture remains unresolved, it may inspire the research community to prove the NP-hardness theoretically. As discussed before, two-ray cases are a subclass of convex-hull cases. If the CTSPP could be proved to be NP-hard on two-ray cases in the future, then it would also prove that the CTSPP on the convex hull is NP-hard.

Table 4.7 provides the number of instances in which the Two-Ray heuristic can obtain equal or better solutions than RIH-$S(3,1)$ for size-30 and -50 two-ray instances. The last two columns show that for all single instances, the solutions generated from the Two-Ray heuristic are equal to or better than those generated from RIH-$S(3,1)$. This shows the ability of the Two-Ray heuristic in obtaining high-quality solutions when all nodes are distributed along two rays.

Table 4.6: Experimental results of Two-Ray heuristic on size-24 two-ray instances: number of instances solved to optimality

Size	Slope	Random or Even group	Solved to optimality
$n = 24$	$s = 1$	Random	10
	$s = 1$	Even	9
	$s = 2$	Random	10
	$s = 2$	Even	10
	$s = 0.5$	Random	9
	$s = 0.5$	Even	10

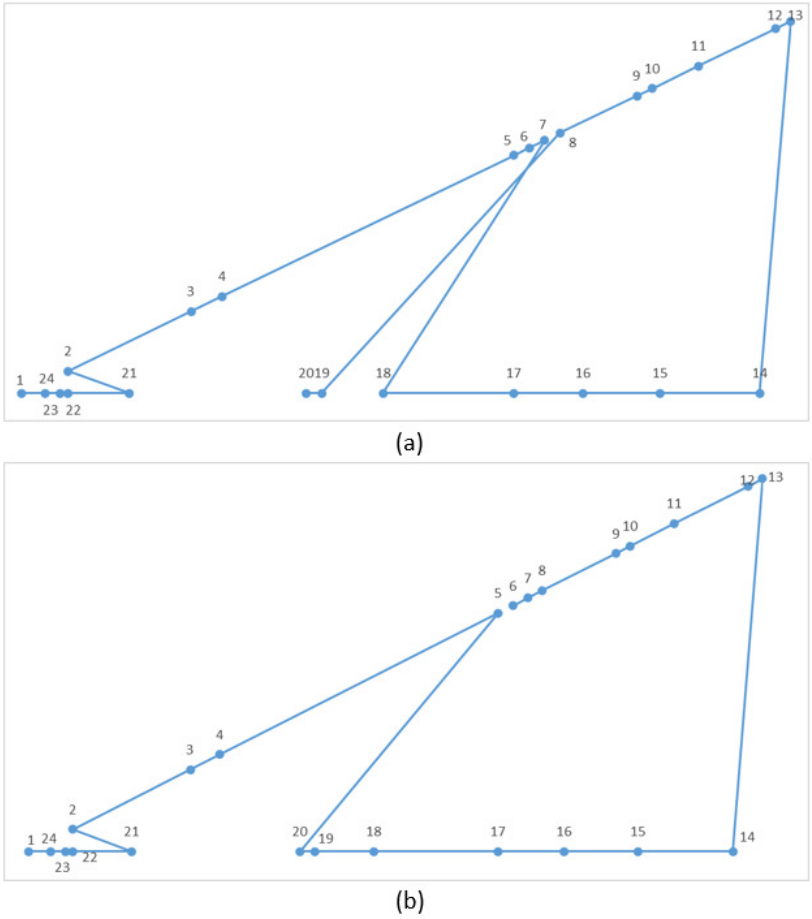


Figure 4.17: Comparison of exact optimal route with Two-Ray heuristic route on 'Difficult-1' case that can be solved by a more sophisticated heuristic: (a) exact optimal route; (b) Two-Ray heuristic route.

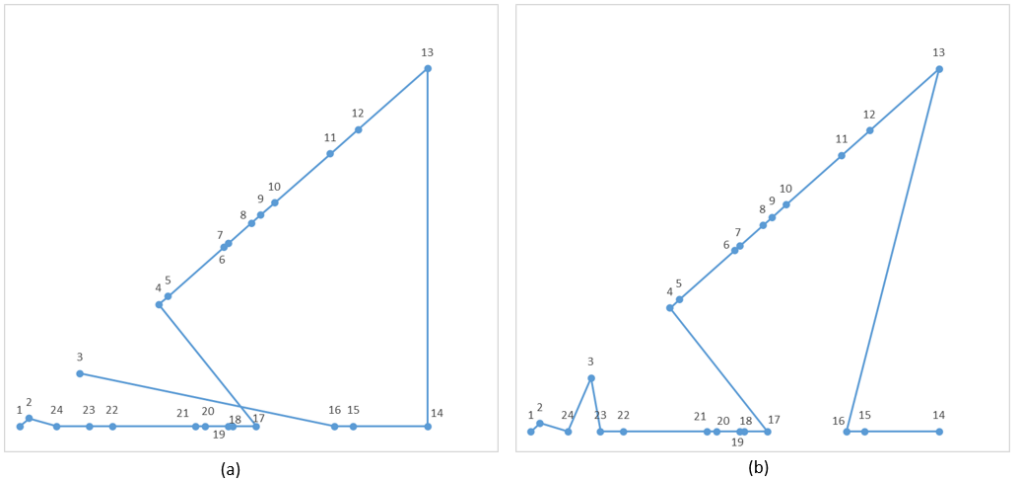


Figure 4.18: Comparison of exact optimal route with Two-Ray heuristic route on 'Difficult-2' case that can be solved by a more sophisticated heuristic: (a) exact optimal route; (b) Two-Ray heuristic route

Table 4.7: Experimental results of Two-Ray heuristic on size-30 and -50 two-ray instances: number of instances improved by Two-Ray heuristic compared with RIH-$S(3,1)$

Size	Slope	Random or Even group	Equal or improved	Improved
$n = 30$	$s = 1$	Random	10	3
	$s = 1$	Even	10	2
	$s = 2$	Random	10	1
	$s = 2$	Even	10	0
	$s = 0.5$	Random	10	1
	$s = 0.5$	Even	10	0
$n = 50$	$s = 1$	Random	10	1
	$s = 1$	Even	10	6
	$s = 2$	Random	10	1
	$s = 2$	Even	10	5
	$s = 0.5$	Random	10	1
	$s = 0.5$	Even	10	2

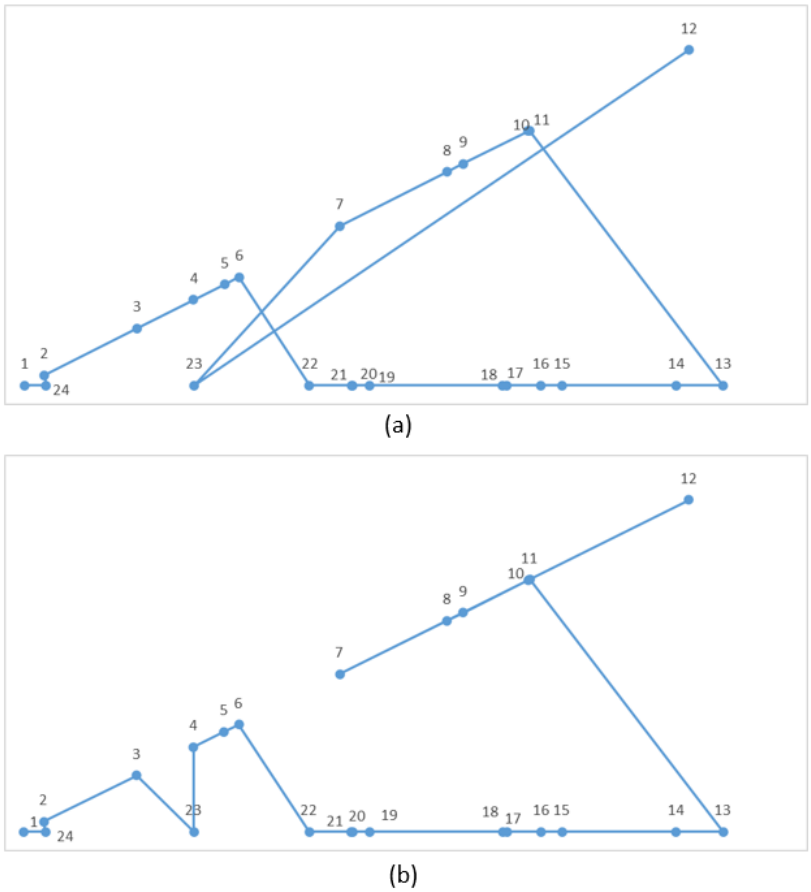


Figure 4.19: Comparison of exact optimal route with Two-Ray heuristic route on a 'More difficult' case that cannot be solved by a more sophisticated heuristic: (a) exact optimal route; (b) Two-Ray heuristic route

4.6 Conclusions and Future Research

In this chapter, we make both theoretical and empirical contributions to the CTSPP on specially structured cases.

In the theory, we extend the theories of Line-CTSPP by fixing the exit node and considering the straight line as part of the total path, where the depot on the line is no longer the first point in the complete path. It is shown that the extended cases can be solved with a dynamic programming algorithm in polynomial time. It is also proved that the more general problem CTSPPW can be solved in polynomial time. In addition, a relationship between the QAP and the CTSPP is discovered, and the time complexity of the CTSPP on the SUM matrix is proved to be $O(nlogn)$. Also, the CTSPP on two rays (a special subclass of the convex hull) is conjectured to be NP-hard.

In the empirical research, we propose dynamic programming heuristics for the CTSPP based on special structures. We conduct computational experiments to show that the proposed heuristics perform well on specially structured cases. The Line heuristic is based on the path structure in the solvable Line-CTSPP and displays superior performance to the classical GRASP-2-opt in both running time and solution quality when the nodes are distributed on two close parallel lines. The Up-Down heuristic is inspired by the Up-Down structure in the solvable Path TSP. This heuristic outperforms the classical GRASP-2-opt in convex-hull cases and close-to-convex-hull cases. Further, the Two-Ray heuristic is proposed for two-ray cases, where all nodes are along two rays. The Two-Ray heuristic combines the Up-Down structure with the Line structure of Line-CTSPP. The experiments suggest that the Two-Ray heuristic can obtain high-quality solutions in the special case that nodes are distributed along two rays.

In the future, we aim to conduct further theoretical research on solvable cases for the CTSPP. The potential cases are those with two rays, two parallel lines and a convex hull. The conjecture encourages us to theoretically prove the NP-hardness of the CTSPP on two rays. In empirical research, heuristics for more general problems can be proposed. Increasing the efficiency of the proposed heuristics discussed in this chapter for a larger-size CTSPP is a further research topic.

Chapter 5

Conclusion and Future Work

5.1 Summary

In this thesis, we have demonstrated how to use solvable cases in heuristics for the TSP and the CTSPP. Solvable cases are not only useful in their own right, but can also improve the performance of heuristics for general cases. This motivated our research on the heuristics inspired by solvability in this thesis. We have proposed heuristics inspired by the solvable cases and demonstrated their promising performance in both theoretical and empirical research. The solvable cases were investigated in three aspects: specially structured matrices that can be solved in polynomial time, exponential neighbourhoods that can be searched in polynomial time and mathematical programming models for small-size problems.

Solvable Specially Structured Matrices

We have made both theoretical and empirical contributions based on the solvable cases, focusing on specially structured matrices. Theoretically, we have found the theoretical property of three classical heuristics, NN, DENN and GREEDY, of obtaining the permutation for permuted strong anti-Robinson matrices for the TSP such as the renumbered matrices satisfy the anti-Robinson conditions . Inspired by the transformation relationship between Kalmanson matrices and anti-Robinson matrices, we have proposed Kalmanson heuristics by making minor amendments to the three classical heuristics. The amended versions have the additional theoretical property of finding the optimal solution for a permuted strong Kalmanson matrix for the TSP. For the CTSPP, we have extended the theories of Line-CTSPP, whose underlying matrix is an anti-Robinson matrix, by fixing the exit node and considering the straight line as part of the total path, and shown

that the extended cases can be solved with a dynamic programming algorithm in polynomial time. We have also proved that the more general problem of CTSPPW can be solved in polynomial time when the nodes are on a line. In addition, we have proved that the time complexity of the CTSPP on the SUM matrix is $O(nlogn)$. Also, we conjecture that the CTSPP on two rays (a special subclass of the convex hull) is NP-hard.

Empirically, the experimental results suggest that our proposed heuristics motivated by specially structured matrices perform well. For the TSP, the empirical investigation on the benchmark instances clearly indicates the superiority of the Kalmanson heuristics to the original counterparts for general cases. The incorporation of additional features from the Kalmanson matrix into the classical heuristics has enriched the nature of the heuristics. For the CTSPP, we have proposed three dynamic programming heuristics based on special structures. The computational experiments have shown that the proposed heuristics have good performance on specially distributed cases. The Line heuristic is based on the path structure in the solvable Line-CTSPP and displays superior performance to the classical GRASP-2-opt in both running time and solution quality when the nodes are distributed on two close parallel lines. The Up-Down heuristic is inspired by the Up-Down structure in the solvable Path TSP and outperforms the classical GRASP-2-opt in convex-hull cases and close-to-convex-hull cases. The Two-Ray heuristic combines the Up-Down structure with the path structure of Line-CTSPP. The experiments suggest that the Two-Ray heuristic can obtain high-quality solutions when nodes are specially distributed along two rays.

Solvable Neighbourhoods

We have developed two tour improvement heuristics for the CTSPP, the pyramidal heuristic and chains heuristic, which can search the pyramidal neighbourhood and chains neighbourhood respectively in polynomial time. The proposed heuristics have simple procedures and better performance than the classical GRASP-2-opt in computationally expensive experiments, especially on specially structured cases of weighted trees.

Solvable Size

The chains heuristic is also motivated by solvable small-size problems. It partitions the tour into a set of chains with small sizes and visits the chains in order. With this heuristic, each chain can be regarded as a small solvable problem. We have proposed another heuristic, the sliding window heuristic, which uses a different strategy to simplify the whole problem to a small solvable problem. This

heuristic arranges intervals to represent new consumers by using a sliding window aggregation strategy. The two heuristics are both based on dynamic programming and outperform the classical GRASP-2-opt for the CTSPP.

In addition to proposing heuristics motivated by solvable cases, we have made an empirical contribution for the CTSPP by comparing the effectiveness of various simple heuristics through extensive computational experiments. The proposed tour construction heuristics IGRASP and RIH have better performance than the widely used GRASP. In addition, the experiments also give general insights into better selecting and combining simple heuristics in the future study of metaheuristics for the CTSPP. The outperformance of the proposed heuristics illustrates their potential of being incorporated into future metaheuristics.

5.2 Future Work

In this section we present directions for future research and extensions of the thesis.

Based on the research in Chapter 2, we can conduct further theoretical research on quantifying how far a given distance matrix is from a Kalmanson matrix and/or anti-Robinson matrix. If we can quantify this, we can further quantify the approximation ratio of the Kalmanson heuristics (with $\alpha = 1$) or we can determine the best value of α to be used.

Chapter 3 gives guidance on the better selection and combination of simple heuristics in the future study of metaheuristics for the CTSPP. In the future, we aim to propose better metaheuristics based on the findings in this chapter. Moreover, the proposed dynamic programming heuristics can be incorporated into future metaheuristics.

In Chapter 4, we conjectured that the CTSPP on two rays is NP-hard. The conjecture motivates us to theoretically prove the NP-hardness in further research. We also aim to conduct theoretical research on other solvable cases for the CTSPP. The potential cases include two parallel lines and convex-hull cases.

Most of the heuristics motivated by the specially structured matrices in this thesis are related to the Kalmanson matrix, which is a subclass of the Demidenko matrix. In future research, we will explore other heuristics using the knowledge of Demidenko matrices for the TSP and the CTSPP. We will also investigate other solvable neighbourhoods and propose heuristics based on them. Increasing the efficiency of our proposed heuristics for the larger-size TSP and CTSPP is another potential research topic. We will also apply our proposed heuristics to more general

problems such as the load-dependent TSP and the cumulative VRP.

Appendix A

Benchmark Instances for TSP

Test instances in the Challenge test suite: Random instances

Random Clustered Euclidean			Random Uniform Euclidean		
Name	HK bound	Optimal	Name	HK bound	Optimal
C1k-0	11325840	11387430	E1k-0	23183212	23360648
C1k-1	11330836	11376735	E1k-1	22839568	22985695
C1k-2	10809149	10855033	E1k-2	22858726	23023351
C1k-3	11823906	11886457	E1k-3	23002034	23143748
C1k-4	11433764	11499958	E1k-4	22542849	22698717
C1k-5	11328719	11394911	E1k-5	23057465	23192391
C1k-6	10092637	10166701	E1k-6	23166620	23349803
C1k-7	10602996	10664660	E1k-7	22666814	22879091
C1k-8	11566102	11605723	E1k-8	22795477	23025754
C1k-9	10835951	10906997	E1k-9	23215285	23356256
C3k-0	19080351	19198258	E3k-0	40348236	40634081
C3k-1	18901572	19017805	E3k-1	40046054	40315287
C3k-2	19410947	19547551	E3k-2	40006528	40303394
C3k-3	19001116	19108508	E3k-3	40318841	40589659
C3k-4	18757585	18864046	E3k-4	40462881	40757209

Test instances in the Challenge test suite: TSPLIB instances

TSPLIB1			TSPLIB2			
Name	HK bound	Optimal	Name	Optimal	Name	Optimal
d1291	50209	50801	a280	2579	p654	34643
d1655	61549	62128	bier127	118282	pcb442	50778
d2103	79307	80450	ch130	6110	pr107	44303
dsj1000	18546977	18660188	ch150	6528	pr124	59030
fl1400	19783	20127	d198	15780	pr136	96772
fl1577	21886	22249	d493	35002	pr144	58537
fl3795	28477	28772	d657	48912	pr152	73682
nrw1379	56396	56638	eil101	629	pr226	80369
pcb1173	56351	56892	fl417	11861	pr264	49135
pcb3038	136588	137694	gil262	2378	pr299	48191
pr1002	256766	259045	kroA100	21282	pr439	107217
pr2392	373490	378032	kroA150	26524	rat195	2323
rl1304	249094	252948	kroA200	29368	rat575	6773
rl1323	265814	270199	kroB100	22141	rat783	8806
rl1889	311704	316536	kroB150	26130	rd100	7910
u1060	222651	224094	kroB200	29437	rd400	15281
u1432	152535	152970	kroC100	20749	ts225	126643
u1817	56688	57201	kroD100	21294	tsp225	3919
u2152	63858	64253	kroE100	22068	u159	42080
u2319	234215	234256	lin105	14379	u574	36905
vm1084	236162	239297	lin318	42029	u724	41910
vm1748	332061	336556	linhp318	41345		

Appendix B

Benchmark Instances for CTSPP

Best known solutions on benchmark instances for CTSPP

Name	n=50	n=100	n=200	n=500
R1	12198	32779	88787	1841386
R2	11621	33435	91977	1816568
R3	12139	32390	92568	1833044
R4	13071	34733	93174	1809266
R5	12126	32598	88737	1823975
R6	12684	34159	91589	1786620
R7	11176	33375	92754	1847999
R8	12910	31780	89048	1820846
R9	13149	34167	86326	1733819
R10	12892	31605	91552	1762741
R11	12103	34188	92655	1797881
R12	10633	32146	91457	1774452
R13	12115	32604	86155	1873699
R14	13117	32433	91882	1799171
R15	11986	32574	88912	1791145
R16	12138	33566	89311	1810188
R17	12176	34198	89089	1825748
R18	13357	31929	93619	1826263
R19	11430	33463	93369	1779248
R20	11935	33632	86292	1820813

Appendix C

'Difficult' Cases on Two Rays for CTSPP

'Difficult-1' case on two rays for CTSPP

Index	X-Coordinate	Y-Coordinate
1	0	0
2	6	3
3	22	11
4	26	13
5	64	32
6	66	33
7	68	34
8	70	35
9	80	40
10	82	41
11	88	44
12	98	49
13	100	50
14	96	0
15	83	0
16	73	0
17	64	0
18	47	0
19	39	0
20	37	0
21	14	0
22	6	0
23	5	0
24	3	0

'Difficult-2' case on two rays for CTSPP

Index	X-Coordinate	Y-Coordinate
1	0	0
2	2	2
3	13	13
4	30	30
5	32	32
6	44	44
7	45	45
8	50	50
9	52	52
10	55	55
11	67	67
12	73	73
13	88	88
14	88	0
15	72	0
16	68	0
17	51	0
18	46	0
19	45	0
20	40	0
21	38	0
22	20	0
23	15	0
24	8	0

'More Difficult' case on two rays for CTSPP

Index	X-Coordinate	Y-Coordinate
1	0.00	0.00
2	31.06	13.83
3	177.23	78.91
4	265.84	118.36
5	314.26	139.92
6	337.10	150.09
7	496.06	220.86
8	664.15	295.70
9	689.73	307.09
10	792.04	352.64
11	793.87	353.45
12	1044.18	464.90
13	1098.00	0.00
14	1024.00	0.00
15	845.00	0.00
16	812.00	0.00
17	758.00	0.00
18	752.00	0.00
19	543.00	0.00
20	516.00	0.00
21	514.00	0.00
22	448.00	0.00
23	267.00	0.00
24	34.00	0.00

Bibliography

[1] Aarts, E., Aarts, E. H., and Lenstra, J. K. (2003). *Local search in combinatorial optimization*. Princeton University Press.

[2] Abeledo, H., Fukasawa, R., Pessoa, A., and Uchoa, E. (2010). The time dependent traveling salesman problem: Polyhedra and branch-cut-and-price algorithm. In *International Symposium on Experimental Algorithms*, pages 202–213. Springer.

[3] Afrati, F., Cosmadakis, S., Papadimitriou, C. H., Papageorgiou, G., and Papakostantinou, N. (1986). The complexity of the travelling repairman problem. *RAIRO-Theoretical Informatics and Applications*, 20(1):79–87.

[4] Aho, A. V. and Hopcroft, J. E. (1974). *The design and analysis of computer algorithms*. Pearson Education India.

[5] Ahuja, R. K., Ergun, O., Orlin, J. B., and Punnen, A. P. (2002). A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics*, 123(1-3):75–102.

[6] Applegate, D. L., Bixby, R. E., Chvatal, V., and Cook, W. J. (2006). *The traveling salesman problem: a computational study*. Princeton university press.

[7] Archer, A. and Blasiak, A. (2010). Improved approximation algorithms for the minimum latency problem via prize-collecting strolls. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete algorithms*, pages 429–447. SIAM.

[8] Archer, A. and Williamson, D. P. (2003). Faster approximation algorithms for the minimum latency problem. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 88–96. Society for Industrial and Applied Mathematics.

[9] Archetti, C. and Speranza, M. G. (2014). A survey on matheuristics for routing problems. *EURO Journal on Computational Optimization*, 2(4):223–246.

[10] Arora, S. and Karakostas, G. (2003). Approximation schemes for minimum latency problems. *SIAM Journal on Computing*, 32(5):1317–1337.

[11] Balas, E. and Christofides, N. (1981). A restricted Lagrangean approach to the traveling salesman problem. *Mathematical Programming*, 21(1):19–46.

[12] Balas, E. and Simonetti, N. (2001). Linear time dynamic-programming algorithms for new classes of restricted TSPs: A computational study. *INFORMS Journal on Computing*, 13(1):56–75.

[13] Balinski, M. L. and Quandt, R. E. (1964). On an integer program for a delivery problem. *Operations Research*, 12(2):300–304.

[14] Ball, M. O. (2011). Heuristics based on mathematical programming. *Surveys in Operations Research and Management Science*, 16(1):21–38.

[15] Barvinok, A., Gimadi, E. K., and Serdyukov, A. I. (2007). The maximum TSP. In *The Traveling Salesman Problem and Its Variations*, pages 585–607. Springer.

[16] Beamon, B. M. (2004). Humanitarian relief chains: issues and challenges. In *Proceedings of the 34th International Conference on Computers and Industrial Engineering*, volume 34, pages 77–82. University of Washington Seattle, WA.

[17] Bentley, J. J. (1992). Fast algorithms for geometric traveling salesman problems. *ORSA Journal on Computing*, 4(4):387–411.

[18] Bianco, L., Mingozzi, A., and Ricciardelli, S. (1993). The traveling salesman problem with cumulative costs. *Networks*, 23(2):81–91.

[19] Bigras, L.-P., Gamache, M., and Savard, G. (2008). The time-dependent traveling salesman problem and single machine scheduling problems with sequence dependent setup times. *Discrete Optimization*, 5(4):685–699.

[20] Blum, A., Chalasani, P., Coppersmith, D., Pulleyblank, B., Raghavan, P., and Sudan, M. (1994). The minimum latency problem. In *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, pages 163–171.

[21] Bock, F. (1958). An algorithm for solving travelling-salesman and related network optimization problems. In *Operations Research*, volume 6, pages 897–897.

[22] Booth, K. S. and Lueker, G. S. (1976). Testing for the consecutive ones property, interval graphs, and graph planarity using pq-tree algorithms. *Journal of Computer and System Sciences*, 13(3):335–379.

[23] Boschetti, M. A., Maniezzo, V., Roffilli, M., and Rohler, A. B. (2009). Matheuristics: Optimization, simulation and control. *Hybrid Metaheuristics*, pages 171–177.

[24] Bowerman, R., Hall, B., and Calamai, P. (1995). A multi-objective optimization approach to urban school bus routing: Formulation and solution method. *Transportation Research Part A: Policy and Practice*, 29(2):107–123.

[25] Brady, R. (1985). Optimization strategies gleaned from biological evolution. *Nature*, 317(6040):804–806.

[26] Burkard, R. E. (1989). Locations with spatial interactions: the quadratic assignment problem. In *Discrtete Location Theory*, pages 387–437. Academic Press.

[27] Burkard, R. E., Deineko, V. G., Van Dal, R., van der Veen, J. A., and Woeginger, G. J. (1998). Well-solvable special cases of the traveling salesman problem: a survey. *SIAM Review*, 40(3):496–546.

[28] Caceres-Cruz, J., Arias, P., Guimarans, D., Riera, D., and Juan, A. A. (2014). Rich vehicle routing problem: Survey. *ACM Computing Surveys (CSUR)*, 47(2):1–28.

[29] Campbell, A. M., Vandenbussche, D., and Hermann, W. (2008). Routing for relief efforts. *Transportation Science*, 42(2):127–145.

[30] Carlier, J. and Villon, P. (1990). A new heuristic for the traveling salesman problem. *RAIRO-Operations Research*, 24(3):245–253.

[31] Carpaneto, G. and Toth, P. (1980). Some new branching and bounding criteria for the asymmetric travelling salesman problem. *Management Science*, 26(7):736–743.

[32] Cela, E. (2013). *The quadratic assignment problem: theory and algorithms*, volume 1. Springer Science & Business Media.

[33] Cela, E., Deineko, V. G., and Woeginger, G. J. (2012). The x-and-y-axes travelling salesman problem. *European Journal of Operational Research*, 223(2):333–345.

[34] Cela, E., Deineko, V. G., and Woeginger, G. J. (2018). New special cases of the quadratic assignment problem with diagonally structured coefficient matrices. *European Journal of Operational Research*, 267(3):818–834.

[35] Cela, E., Deineko, V. G., and Woeginger, G. J. (2021). Travelling salesman paths on Demidenko matrices. *Discrete Applied Mathematics*.

[36] Chalasani, P., Motwani, R., and Rao, A. (1997). Algorithms for robot grasp and delivery. In *Proceeding 2nd international workshop on algorithmic foundations of robotics (WAFR), Toulouse, France*, page 347.

[37] Chaudhuri, K., Godfrey, B., Rao, S., and Talwar, K. (2003). Paths, trees, and minimum latency tours. In *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.*, pages 36–45. IEEE.

[38] Christofides, N. (1970). The shortest hamiltonian chain of a graph. *SIAM Journal on Applied Mathematics*, 19(4):689–696.

[39] Christofides, N., Mingozzi, A., and Toth, P. (1981). State-space relaxation procedures for the computation of bounds to routing problems. *Networks*, 11(2):145–164.

[40] Christopher, G., Farach, M., and Trick, M. (1996). The structure of circular decomposable metrics. In *European Symposium on Algorithms*, pages 486–500. Springer.

[41] Clarke, G. and Wright, J. W. (1964). Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research*, 12(4):568–581.

[42] Coelho, L. C., Renaud, J., and Laporte, G. (2016). Road-based goods transportation: a survey of real-world logistics applications from 2000 to 2015. *INFOR: Information Systems and Operational Research*, 54(2):79–96.

[43] Coene, S., Filippi, C., Spieksma, F. C., and Stevanato, E. (2013). Balancing profits and costs on trees. *Networks*, 61(3):200–211.

[44] Croes, G. A. (1958). A method for solving traveling-salesman problems. *Operations Research*, 6(6):791–812.

[45] Dantzig, G., Fulkerson, R., and Johnson, S. (1954). Solution of a large-scale traveling-salesman problem. *Journal of the Operations Research Society of America*, 2(4):393–410.

[46] Dantzig, G. B. and Ramser, J. H. (1959). The truck dispatching problem. *Management science*, 6(1):80–91.

[47] Davendra, D. (2010). *Traveling Salesman Problem: Theory and Applications*. BoD–Books on Demand.

[48] Deineko, V. G. and Filonenko, V. (1979). On the reconstruction of specially structured matrices. *Aktualnyje Problemy EVM, programmirovanije, Dnepropetrovsk, DGU (in Russian)*.

[49] Deineko, V. G., Klinz, B., and Wang, M. (2022). 2-period balanced travelling salesman problem: a polynomially solvable case and heuristics. *arXiv preprint arXiv:2203.06090*.

[50] Deineko, V. G., Rudolf, R., and Woeginger, G. J. (1996). On the recognition of permuted Supnick and incomplete Monge matrices. *Acta Informatica*, 33(5):559–569.

[51] Deineko, V. G. and Woeginger, G. (2000a). A study of exponential neighborhoods for the travelling salesman problem and for the quadratic assignment problem. *Mathematical Programming*, 87(3):519–542.

[52] Deineko, V. G. and Woeginger, G. J. (1998). A solvable case of the quadratic assignment problem. *Operations Research Letters*, 22(1):13–17.

[53] Deineko, V. G. and Woeginger, G. J. (2000b). The maximum travelling salesman problem on symmetric Demidenko matrices. *Discrete Applied Mathematics*, 99(1-3):413–425.

[54] Demidenko, V. (1976). A special case of traveling salesman problems. *Izv. Akad. Nauk. BSSR, Ser. Fiz.-mat. Nauk*, 5:28–32 (in Russian).

[55] Deuineko, V. G., Rudolf, R., and Woeginger, G. J. (1998). Sometimes travelling is easy: the master tour problem. *SIAM Journal on Discrete Mathematics*, 11(1):81–93.

[56] Dror, M. and Trudeau, P. (1989). Savings by split delivery routing. *Transportation Science*, 23(2):141–145.

[57] Elhallaoui, I., Villeneuve, D., Soumis, F., and Desaulniers, G. (2005). Dynamic aggregation of set-partitioning constraints in column generation. *Operations Research*, 53(4):632–645.

[58] Erickson, J. (1999). Algorithms.

[59] Ezzine, I. O., Semet, F., and Chabchoub, H. (2010). New formulations for the traveling repairman problem. In *Proceedings of the 8th International conference of modeling and simulation*. Citeseer.

[60] Fakcharoenphol, J., Harrelson, C., and Rao, S. (2007). The k-traveling repairmen problem. *ACM Transactions on Algorithms (TALG)*, 3(4):40–es.

[61] Feillet, D., Dejax, P., and Gendreau, M. (2005). Traveling salesman problems with profits. *Transportation Science*, 39(2):188–205.

[62] Feo, T. A. and Resende, M. G. (1995). Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6(2):109–133.

[63] Fischetti, M., Laporte, G., and Martello, S. (1993). The delivery man problem and cumulative matroids. *Operations Research*, 41(6):1055–1064.

[64] Fisher, M. L. and Jaikumar, R. (1981). A generalized assignment heuristic for vehicle routing. *Networks*, 11(2):109–124.

[65] Flood, M. M. (1956). The traveling-salesman problem. *Operations Research*, 4(1):61–75.

[66] Frieze, A. (1979). Worst-case analysis of algorithms for travelling salesman problems. *Methods of Operations Research*, 32:97–112.

[67] Garcia, A. and Tejel, J. (1996). Using total monotonicity for two optimization problems on the plane. *Information Processing Letters*, 60(1):13–17.

[68] Gendreau, M., Hertz, A., and Laporte, G. (1992). New insertion and postoptimization procedures for the traveling salesman problem. *Operations Research*, 40(6):1086–1094.

[69] Gendreau, M., Laporte, G., and Seguin, R. (1996). Stochastic vehicle routing. *European Journal of Operational Research*, 88(1):3–12.

[70] Gendreau, M. and Potvin, J.-Y. (1998). Dynamic vehicle routing and dispatching. In *Fleet Management and Logistics*, pages 115–126. Springer.

[71] Gillett, B. E. and Miller, L. R. (1974). A heuristic algorithm for the vehicle-dispatch problem. *Operations Research*, 22(2):340–349.

[72] Gilmore, P., Lawler, E., and Shmoys, D. (1985). The traveling salesman problem, chapter 4 well-solved special cases.

[73] Glover, F. (1977). Heuristics for integer programming using surrogate constraints. *Decision Sciences*, 8(1):156–166.

[74] Glover, F. (1986). Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, 13(5):533–549.

[75] Glover, F., Gutin, G., Yeo, A., and Zverovich, A. (1999). Construction heuristics and domination analysis for the asymmetric TSP. In *International Workshop on Algorithm Engineering*, pages 85–94. Springer.

[76] Goetschalckx, M. and Jacobs-Blecha, C. (1989). The vehicle routing problem with backhauls. *European Journal of Operational Research*, 42(1):39–51.

[77] Golden, B. L. (1985). Empirical analysis of heuristics. In *The Traveling Salesman Problem*, pages 218–222. Wiley.

[78] Golden, B. L., Raghavan, S., Wasil, E. A., et al. (2008). *The vehicle routing problem: latest advances and new challenges*, volume 43. Springer.

[79] Gu, J. and Huang, X. (1994). Efficient local search with search space smoothing: A case study of the traveling salesman problem (TSP). *IEEE Transactions on Systems, Man, and Cybernetics*, 24(5):728–735.

[80] Gutin, G. (1999). Exponential neighbourhood local search for the traveling salesman problem. *Computers & Operations Research*, 26(4):313–320.

[81] Gutin, G. and Punnen, A. P. (2006). *The traveling salesman problem and its variations*, volume 12. Springer Science & Business Media.

[82] Gutin, G., Yeo, A., and Zverovich, A. (2002). Exponential neighborhoods and domination analysis for the tsp. In *The Traveling Salesman Problem and Its Variations*. Kluwer, Dordrecht.

[83] Hardy, G. H., Littlewood, J. E., Polya, G., Littlewood, D., Polya, G., et al. (1952). *Inequalities*. Cambridge university press.

[84] Hasle, G. and Kloster, O. (2007). Industrial vehicle routing. In *Geometric Modelling, Numerical Simulation, and Optimization*, pages 397–435. Springer.

[85] Held, M. and Karp, R. M. (1971). The traveling-salesman problem and minimum spanning trees: Part ii. *Mathematical Programming*, 1(1):6–25.

[86] Hoogeveen, J. (1991). Analysis of christofides' heuristic: Some paths are more difficult than cycles. *Operations Research Letters*, 10(5):291–295.

[87] Hougardy, S. and Wilde, M. (2015). On the nearest neighbor rule for the metric traveling salesman problem. *Discrete Applied Mathematics*, 195:101–103.

[88] Hurkens, C. A. and Woeginger, G. J. (2004). On the nearest neighbor rule for the traveling salesman problem. *Operations Research Letters*, 32(1):1–4.

[89] Johnson, D. S. and McGeoch, L. A. (1997). The traveling salesman problem: A case study in local optimization. *Local Search in Combinatorial Optimization*, 1(1):215–310.

[90] Johnson, D. S. and McGeoch, L. A. (2007). Experimental analysis of heuristics for the STSP. In *The Traveling Salesman Problem and Its Variations*, pages 369–443. Springer.

[91] Johnson, D. S., McGeoch, L. A., and Rothberg, E. E. (1996). Asymptotic experimental analysis for the held-karp traveling salesman bound. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, volume 341, page 350. ACM Press San Francisco.

[92] Kalmanson, K. (1975). Edgeconvex circuits and the traveling salesman problem. *Canadian Journal of Mathematics*, 27(5):1000–1010.

[93] Karp, R. M. (1979). A patching algorithm for the nonsymmetric traveling-salesman problem. *SIAM Journal on Computing*, 8(4):561–573.

[94] Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, 220(4598):671–680.

[95] Klinz, B. and Woeginger, G. J. (1999). The Steiner tree problem in Kalmanson matrices and in circulant matrices. *Journal of Combinatorial Optimization*, 3(1):51–58.

[96] Kontoravdis, G. and Bard, J. F. (1995). A GRASP for the vehicle routing problem with time windows. *ORSA Journal on Computing*, 7(1):10–23.

[97] Koopmans, T. C. and Beckmann, M. (1957). Assignment problems and the location of economic activities. *Econometrica: Journal of the Econometric Society*, pages 53–76.

[98] Koutsoupias, E., Papadimitriou, C., and Yannakakis, M. (1996). Searching a fixed graph. In *International Colloquium on Automata, Languages, and Programming*, pages 280–289. Springer.

[99] Langevin, A., Soumis, F., and Desrosiers, J. (1990). Classification of travelling salesman problem formulations. *Operations Research Letters*, 9(2):127–132.

[100] Laporte, G., Gendreau, M., Potvin, J.-Y., and Semet, F. (2000). Classical and modern heuristics for the vehicle routing problem. *International Transactions in Operational Research*, 7(4-5):285–300.

[101] Laporte, G., Nobert, Y., and Taillefer, S. (1988). Solving a family of multi-depot vehicle routing and location-routing problems. *Transportation Science*, 22(3):161–172.

[102] Lawler, E. (1995). The quadratic assignment problem: a brief review. In *Combinatorial Programming: Methods and Applications*, pages 351–360. Springer.

[103] Lawler, E. L. (1963). The quadratic assignment problem. *Management Science*, 9(4):586–599.

[104] Lawler, E. L. (1985). The traveling salesman problem: a guided tour of combinatorial optimization. *Wiley-Interscience Series in Discrete Mathematics*.

[105] Lin, S. (1965). Computer solutions of the traveling salesman problem. *Bell System Technical Journal*, 44(10):2245–2269.

[106] Lin, S. and Kernighan, B. W. (1973). An effective heuristic algorithm for the traveling-salesman problem. *Operations Research*, 21(2):498–516.

[107] Lucena, A. (1990). Time-dependent traveling salesman problem–the deliveryman case. *Networks*, 20(6):753–763.

[108] Mendez-Diaz, I., Zabala, P., and Lucena, A. (2008). A new formulation for the traveling deliveryman problem. *Discrete Applied Mathematics*, 156(17):3223–3237.

[109] Miller, C. E., Tucker, A. W., and Zemlin, R. A. (1960). Integer programming formulation of traveling salesman problems. *Journal of the ACM (JACM)*, 7(4):326–329.

[110] Miller, D. L. and Pekny, J. F. (1991). Exact solution of large asymmetric traveling salesman problems. *Science*, 251(4995):754–761.

[111] Minieka, E. (1989). The delivery man problem on a tree network. *Annals of Operations Research*, 18(1):261–266.

[112] Mladenovic, N. and Hansen, P. (1997). Variable neighborhood search. *Computers & Operations Research*, 24(11):1097–1100.

[113] Mladenovic, N., Urovsevic, D., and Hanafi, S. (2013). Variable neighborhood search for the travelling deliveryman problem. *4OR*, 11(1):57–73.

[114] Moshref-Javadi, M. and Lee, S. (2013). A taxonomy to the class of minimum latency problems. In *IIE Annual Conference. Proceedings*, page 3896. Institute of Industrial and Systems Engineers (IISE).

[115] Newton, R. M. and Thomas, W. H. (1974). Bus routing in a multi-school system. *Computers & Operations Research*, 1(2):213–222.

[116] Norback, J. P. and Love, R. F. (1977). Geometric approaches to solving the traveling salesman problem. *Management Science*, 23(11):1208–1223.

[117] Oliver, I., Smith, D., and Holland, J. R. (1987). Study of permutation crossover operators on the traveling salesman problem. In *Genetic algorithms and their applications: proceedings of the second International Conference on Genetic Algorithms: July 28-31, 1987 at the Massachusetts Institute of Technology, Cambridge, MA*. Hillsdale, NJ: L. Erlhaum Associates.

[118] Ong, H. L. and Moore, J. (1984). Worst-case analysis of two travelling salesman heuristics. *Operations Research Letters*, 2(6):273–277.

[119] Or, I. (1976). *Travelling Salesman Type Combinatorial Problems and Their Relation to the Logistics of Regional Bolld Banking*. Northwestern University.

[120] Orlin, J. B. and Sharma, D. (2004). Extended neighborhood: Definition and characterization. *Mathematical Programming*, 101(3):537–559.

[121] Papadimitrou, C. H. and Steiglitz, K. (1982). Combinatorial optimization: algorithms and complexity.

[122] Polyakovskiy, S., Spieksma, F. C., and Woeginger, G. J. (2013). The three-dimensional matching problem in Kalmanson matrices. *Journal of Combinatorial Optimization*, 26(1):1–9.

[123] Prea, P. and Fortin, D. (2014). An optimal algorithm to recognize robinsonian dissimilarities. *Journal of Classification*, 31(3):351–385.

[124] Prufer, H. (1918). Neuer bewis eines satzes uber permutationnen. *Arch. Math. Phys.*, 27:742–744.

[125] Punnen, A. and Kabadi, S. (2002). Domination analysis of some heuristics for the traveling salesman problem. *Discrete Applied Mathematics*, 119(1-2):117–128.

[126] Punnen, A., Margot, F., and Kabadi, S. (2003). TSP heuristics: domination analysis and complexity. *Algorithmica*, 35(2):111–127.

[127] Punnen, A. P. (2001). The traveling salesman problem: new polynomial approximation algorithms and domination analysis. *Journal of Information and Optimization Sciences*, 22(1):191–206.

[128] Reinelt, G. (1992). Fast heuristics for large geometric traveling salesman problems. *ORSA Journal on Computing*, 4(2):206–217.

[129] Reinelt, G. (1994). *The traveling salesman: computational solutions for TSP applications*. Springer-Verlag.

[130] Rendl, F., Pardalos, P., and Wolkowicz, H. (1994). The quadratic assignment problem: A survey and recent developments. In *Proceedings of the DIMACS workshop on quadratic assignment problems*, volume 16, pages 1–42.

[131] Robinson, W. S. (1951). A method for chronologically ordering archaeological deposits. *American Antiquity*, 16(4):293–301.

[132] Rodrigue, J.-P. (2020). *The geography of transport systems*. Routledge.

[133] Rosenkrantz, D. J., Stearns, R. E., and Lewis, II, P. M. (1977). An analysis of several heuristics for the traveling salesman problem. *SIAM Journal on Computing*, 6(3):563–581.

[134] Salehipour, A., Sorensen, K., Goos, P., and Braysy, O. (2011). Efficient GRASP+ VND and GRASP+ VNS metaheuristics for the traveling repairman problem. *4OR*, 9(2):189–209.

[135] Sarvanov, V. and Doroshko, N. (1981). Approximate solution of the traveling salesman problem by a local algorithm with scanning neighbourhoods of factorial cardinality in cubic time. *Software: Algorithms and Programs*, 31:8–13 (in Russian).

[136] Savelsbergh, M. W. (1985). Local search in routing problems with time windows. *Annals of Operations Research*, 4(1):285–305.

[137] Silva, M. M., Subramanian, A., Vidal, T., and Ochi, L. S. (2012). A simple and effective metaheuristic for the minimum latency problem. *European Journal of Operational Research*, 221(3):513–520.

[138] Sitters, R. (2002). The minimum latency problem is NP-hard for weighted trees. In *International conference on integer programming and combinatorial optimization*, pages 230–239. Springer.

[139] Stewart, W. R. (1977). A computationally efficient heuristic for the traveling salesman problem. In *Proceedings of the 13th Annual Meeting of Southeastern TIMS, Myrtle Beach, SC, USA*, pages 75–83.

[140] Supnick, F. (1957). Extreme hamiltonian lines. *Annals of Mathematics*, pages 179–201.

[141] Toth, P. and Vigo, D. (2014). *Vehicle routing: problems, methods, and applications*. SIAM.

[142] Tsitsiklis, J. N. (1992). Special cases of traveling salesman and repairman problems with time windows. *Networks*, 22(3):263–282.

[143] Valtr, P. (1994). Probability that n random points are in convex position.

[144] Van der Veen, J. A. (1994). A new class of pyramidally solvable symmetric traveling salesman problems. *SIAM Journal on Discrete Mathematics*, 7(4):585–592.

[145] Van Hentenryck, P. (1989). *Constraint satisfaction in logic programming*. MIT press.

[146] Vitasek, K. (2009). Supply chain management terms and glossary. *Bellevue, Washington: Supply Chain Visions*.

[147] Wu, B. Y. (2000). Polynomial time algorithms for some minimum latency problems. *Information Processing Letters*, 75(5):225–229.

[148] Wu, B. Y., Huang, Z.-N., and Zhan, F.-J. (2004). Exact algorithms for the minimum latency problem. *Information Processing Letters*, 92(6):303–309.

[149] Zenklusen, R. (2019). A 1.5-approximation for path TSP. In *Proceedings of the thirtieth annual ACM-SIAM symposium on discrete algorithms*, pages 1539–1549. SIAM.