2002

Solving traveling salesman problems with heuristic learning approach

Sim Kim Lau
University of Wollongong

SOLVING TRAVELING SALESMAN PROBLEMS WITH HEURISTIC LEARNING APPROACH

A thesis submitted in partial fulfilment of the requirements for the award of the degree

DOCTOR OF PHILOSOPHY

from

THE UNIVERSITY OF WOLLONGONG

by

SIM KIM LAU
MBus(IT), BSc(Hons)

DEPARTMENT OF INFORMATION SYSTEMS
2002

DECLARATION

This is to certify that the work presented in this thesis was carried out by the author at the Department of Information Systems at the University of Wollongong, and is the result of original research and has not been submitted for a degree at any other university or institution.

Sim Kim Lau

ABSTRACT

This research studies the feasibility of applying heuristic learning algorithm in artificial intelligence to address the traveling salesman problem. The research focuses on tour construction and seeks to overcome the weakness of traditional tour construction heuristics, which are of greedy and myopic nature. The advantage of tour construction heuristic is its simplicity in concept. However, the greedy and myopic nature of tour construction heuristic result in a sub-optimal solution, and the tour needs to be improved with much effort after it is built. The improvement is made using tour improvement heuristics, which improves tour by changing the tour configuration until a better solution is found. Traditional tour construction heuristics were not designed to modify the configuration of the tour, which is an important feature in tour improvement heuristics, during the tour construction process. This research investigates the application of a real time admissible heuristic learning algorithm that allows the tour configuration to change as the tour is built. The heuristic evaluation function of the algorithm considers both local and global estimated distance information. The search engine of the algorithm incorporates Delaunay triangulations of computational geometry as part of the search strategy. This helps to improve the search efficiency because computational geometry provides information about the geometric properties of nodes distributed in Euclidean plane, so that only promising nodes that are likely to be in the optimal tour are considered during the search process.

A state space transformation process that defines state, state transition operator and state transition cost has been developed. The heuristic estimation of a state is computed using minimal spanning tree, and the set of relevant states for consideration at each state selection is identified through the application of Delaunay triangulations. Computational results show that the geometric distribution of nodes in the Euclidean plane influences the heuristic estimation, because it influences the computation of minimal spanning tree. Problems that exhibit distinct and well-separated clusters are advantageous to this approach because it is capable of producing good quality heuristic estimate.

A restrictive search approach that could further reduce the search space of the heuristic learning algorithm has also been investigated. It is based on the characteristics of optimal tour in which nodes located on the convex hull are visited in the order in which they appear on the convex hull boundary. Using this characteristic together with the proximity concept of Voronoi diagram in computational geometry, some of the nodes that are unlikely to travel in the same direction as the optimal tour can be pruned. This approach could identify promising candidate edge set using only edges from one triangle selected from Delaunay triangulations, and the triangle is selected based on the direction the tour travels. The examples used in this research show that the saving in heuristic updates can be quite significant.

ACKNOWLEDGEMENTS

I wish to acknowledge the assistance and support of the following people in completing this thesis.

I am indebted to Professor Li-Yen Shue who is now with the National Kaoshiung First University of Science & Technology, Taiwan, without whom this thesis would never have been finished. His supervision and guidance over the years have helped me to persevere in this research direction, and his untiring efforts in pointing me to the right research direction and to teach me to be a good researcher. Most importantly he has continued to support and guide me even after his departure from the Department.

I am also grateful to Professor Graham Winley, Dr Reza Zamani and Mr Louie Athanasiadis, who have provided valuable advice and assistance, and especially to Louie who has assisted in providing programming guidance.

Finally, to my husband and two young daughters who have to tolerate my long working hours while I embarked on the final stage of completing the thesis.

I also acknowledge the use of LEDA libraries and Concorde program in this research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF PUBLICATIONS

Lau, S.K. "Heuristic learning approach for travelling salesman problems", submitted to The Sixteenth Triennial Conference of the International Federation of Operational Research Societies (IFORS 2002), Edinburgh, 8 - 12 July, 2002

Lau, S.K. and Shue, L.Y. 2001. "Solving Travelling Salesman Problems with an Intelligent Search Approach", *Asia-Pacific Journal of Operational Research*, 18(1), 77-87

Lau, S.K. and Shue, L.Y., 2000, "Solving Traveling Salesman Problems with an Intelligent Search Algorithm", *Proceedings of the Fifth Conference of the Association of Asian-Pacific Operations Research Societies (CD-ROM),* July 5-7, Singapore, 7pp.

Lau, S.K. and Shue, L.Y., 2000, "A Heuristic Learning Algorithm for Traveling Salesman Problems", *Proceedings of the Americas Conference on Information Systems,* August 10-13 Long Beach, California, 17-19

CHAPTER 1: INTRODUCTION

This chapter provides an outline of the thesis. The chapter is organised as follows. Section 1 presents the statement of the problem. The research objectives are outlined in Section 2 and an overview of the research follows in Section 3. Section 4 outlines the organisation of this thesis.

1.1 Statement of the problem

In the traveling salesman problem (TSP), a salesman is to find the shortest tour of a finite number of cities by visiting each city exactly once and returning to the starting city. It provides an ideal platform for the study of combinatorial optimisation problems because many industrial optimisation problems can be formulated as TSP. Examples of such applications include vehicle routing, workshop scheduling, order-picking in a warehouse, computer wiring and drilling of circuit board problems.

TSP is inherently intractable. It belongs to a group of problems known as NP-complete. The combinatorial nature of the problem results in computational time to grow exponentially with problem size, and no efficient algorithm could be constructed to find optimal solution in polynomial time for problems that are NP-complete (Garey and Johnson, 1979). Therefore, researchers usually solve the problem by finding approximate solution with reasonable computation time. The common approach is to first construct the tour using tour construction heuristics and then tour improvement heuristic is applied to obtain a better solution. Tour construction heuristic is simple in concept. It works by adding city one at a time using some selection and insertion

criteria. Then, tour improvement heuristic is applied until a shorter tour is found. Tour improvement heuristic works by exchanging edges of the tour. This method allows tour configuration to change during the iterative tour improvement process. However, it can result in local optimal solution. Extensive research has been conducted to address this problem, and a number of global optimisation techniques, such as simulated annealing, genetic algorithm and tabu search, have been developed for this purpose. In contrast, little attempt has been made to construct an optimal tour in the first place, so that tour improvement heuristic needs not be applied after the tour is built. In addition, little research has been conducted to utilise the advancement in the area of artificial intelligence, in particular the heuristic learning principle, to address the TSP.

Tour construction heuristic by itself is not useful because the solution is sub-optimal. It is a greedy approach. Part of the tour that is already built remains unchanged throughout the tour construction process, and no attempt is made to change the tour configuration as the tour is built. This characteristic is in contrast to the tour improvement heuristic which changes the configuration of the tour during the iterative improvement process until a shorter tour is found. In addition, tour construction heuristic is myopic. It often relies on local knowledge to construct a tour. The selection and insertion criteria in various tour construction heuristics rely on local distance information to determine which city is to be selected and added to the tour. Therefore, if the configuration of the tour can be changed during the tour construction process, similar to the approach of tour improvement heuristics, then it is more likely to result in optimal solution.

The aim of this research is to demonstrate the well-developed heuristic learning algorithm in artificial intelligence may present another approach in addressing TSP. A heuristic learning algorithm is applied so that a dynamic tour construction process that allows the tour configuration to change during the tour construction process can be developed. As the tour is constructed, the tour configuration changes through the heuristic learning process using the local and estimated global distance information of the tour. The heuristic learning process allows the heuristic estimates of the partially completed tour to be updated, and, at the same time, the forward search and backtracking operations in the algorithm allow addition and deletion of cities to and from the tour during the tour construction process.

The combinatorial nature of the problem can result in the solution space to become exponential in relation to the problem size. Therefore, it is important to control the search space. An efficient search strategy that can reduce the search space is very important in this research. This research seeks to investigate a search strategy by exploiting the geometric properties of the cities. Computational geometry concepts of Delaunay triangulations and Voronoi diagram will be investigated for this purpose because they provide information about location and neighbourhood of nodes in the Euclidean plane.

1.2 Objectives

This research has three objectives.

1. To investigate how the well-developed heuristic learning algorithm in artificial intelligence can be applied as an approach in addressing the traditional TSP.

2. To develop an approach to implement the heuristic learning algorithm so that TSP can be solved using dynamic tour construction.

3. To demonstrate the computational geometric properties of Euclidean TSP can be utilised as part of the search strategy to reduce the search space.

To achieve these objectives, the research addresses the following steps:

1. Development of a transformation method that can facilitate the formulation of TSP into state-space problems.

2. Development of an approach that incorporates computational geometry of Delaunay triangulation into the search process.

3. Development of a step-by-step application procedure that incorporates heuristic learning approach with Delaunay triangulation as a search strategy.

4. Investigation of factors that affect the performance of the proposed heuristic learning approach.

5. Development of a restrictive search approach through the integration of knowledge with regard to the direction of the tour and the computational geometric property of Voronoi diagram in the heuristic learning approach.

1.3 Overview of the research

This research investigates the application of a heuristic learning algorithm called Search and Learning A* algorithm (SLA*) to solve TSP. SLA* is a real time admissible heuristic learning algorithm, which uses the heuristic evaluation function to

estimate the relative merit of different state to the goal state (Zamani, 1995). The heuristic estimate of a state represents an estimate of solution from that state to the goal state. The rationality of this algorithm is that, a state that is further away from the goal state should have a larger heuristic estimate. The feature of this algorithm is the application of heuristic learning principle to update and improve the initially underestimated heuristic estimates. This will lead to an improvement of state selection decision and an optimal solution when the goal state is reached.

In order to apply the heuristic learning approach to overcome the greedy and myopic nature of the tour construction heuristics, TSP is transformed into a state-space problem. The heuristic evaluation function of the algorithm considers both local and estimated global distances. Local distance is the actual cost of moving from one state to another, and estimated global distance is the heuristic estimation of a state to the goal state. The heuristic learning mechanism allows the algorithm to update the heuristic estimates of the visited states, and hence modify the tour configuration along the search process. This way the tour configuration changes as a result of heuristic learning by utilising local and global distance information during the tour construction process.

Search efficiency of the heuristic learning approach can be improved by considering only those edges that are likely to lead to an optimal tour. In order to reduce search space, the concept of Delaunay triangulation is used to construct candidate edge set in which only promising edges are selected during the search process. In addition, the concept of proximity using Voronoi diagram, and the direction of the tour are integrated in the search strategy to further reduce the search space. A learning threshold method will also be investigated to find approximate solution. This method

improves the computation time for large-sized problems at the sacrifice of the quality of the solution.

1.4 Outline of the thesis

The rest of the thesis is organised as follows.

Chapter 2 presents the literature review, which covers various tour construction and tour improvement heuristics that are commonly used to solve TSP. Examples of tour construction heuristics examined include nearest neighbour heuristics, insertion heuristics and Christofides' heuristics. The tour improvement heuristics investigates the local search approach as well as global optimisation techniques, which include simulated annealing, genetic algorithm and tabu search. The branch and bound method that has been successfully applied to solve large TSP will also be explored.

Chapter 3 examines the tour construction process using the heuristic learning approach of SLA*. This chapter investigates the state-space transformation process to transform TSP into a state-space problem. The transformation process includes definitions of state, state transition operator, state transition cost and heuristic estimates. The concept of Delaunay triangulation will be examined, and is incorporated as part of the search strategy to find promising neighbouring cities in the tour construction process. The step-by-step application procedure to construct tour using the heuristic learning approach will be given. This is followed by an example to demonstrate the working of the algorithm. Finally, an approximation method using the principle of learning threshold will be examined. This is followed by three examples to demonstrate the step-by-step procedure of the algorithm.

Chapter 4 investigates the implementation of the heuristic learning approach to construct tour. Issues relating to computer implementation of the algorithm are discussed. The computational experiements were conducted on two sets of test problems: selected instances from the TSPLIB library and the randomly generated problems. Factors influencing the performance of the heuristic learning approach will be investigated and discussed.

Chapter 5 examines the development of a restrictive search strategy. This chapter investigates the rationale behind the implementation of the restrictive search approach, which is used as a constrained search strategy to further reduce the search space. The factors considered are based on the concept of proximity in Voronoi diagram, direction of the tour and the search direction of a triangle from Delaunay triangulations. Each of the factors will be investigated. Three examples have been included to demonstrate the implementation of the restrictive search approach.

Chapter 6 concludes the thesis with the discussion of possible future research direction.

CHAPTER 2: LITERATURE REVIEW

2.1 Introduction

The nature of NP-completeness in TSP makes it unlikely that any algorithm can be guaranteed to find optimal solutions when the number of cities is large. In view of the computation difficulties various heuristics have been developed to solve the problem. This chapter presents a literature review of various heuristics methods. The literature review covers commonly used tour construction and tour improvement heuristics. In addition a number of global optimisation techniques, which can be used to overcome the local entrapment problems in local search heuristics, will also be examined. The branch-and-bound and branch-and-cut methods that have been successfully used to solve large TSP are also examined.

This chapter is organised as follows: Section 2 discusses tour construction heuristics. This includes nearest neighbour, various insertion heuristics and Christofides' heuristic. Other tour construction methods such as space filling curve and neural network will also be outlined. Section 3 examines tour improvement heuristics. This includes local search heuristic as well as global optimisation heuristics that include simulated annealing, genetic algorithm and tabu search. The algorithm of each of these techniques and its associated advantages, disadvantages and performances will be discussed. Section 4 examines the application of the branch-and-bound and branch-and-cut methods to solve large TSP. The conclusion follows in Section 4.

2.2 Tour construction heuristics

Tour construction heuristic constructs a tour by progressively adding a city one at a time until a complete tour is formed. The part of the tour that is already built remains unchanged during the tour construction process. In this section, nearest neighbour heuristic, various insertion heuristics, Christofides' heuristic, space filling curve heuristic and neural network approach will be examined.

2.2.1 Nearest neighbour heuristic

Nearest neighbour heuristic is simple in concept. It randomly selects a city as the starting node of the path. The next city for inclusion is the unvisited city that is nearest to the last city. The process is repeated until all cities have been included. Finally the last city is joined to the first city to form a tour.

The running time of this heuristic is found to be proportional to n^2, represented as $O(n^2)$ where n is the number of city (Golden et al, 1980). The expected tour length is of the order O(logn) times the optimal tour length for random distance problem (Johnson, 1990). The quality of the solution is strongly dependent on the choice of the starting city (Reinelt, 1994). One way to overcome this problem is to repeat the nearest neighbour heuristic for each possible starting city and the tour with the shortest distance is selected as the optimal tour (Rosenkrantz et al, 1977). But the running time using this approach is proportional to n^3.

Although the nearest neighbour heuristic is simple in concept, it has its weakness. By being greedy in the beginning stage of the tour construction process, the tour distance may increase considerably in length when the last city is joined to the first city. A

number of researchers have modified nearest neighbour heuristic to overcome this shortcoming. Burke (1994) uses a tour, instead of a path, during the construction process. Bentley (1992) uses double-ended nearest neighbour approach that allows the path to grow at both ends of the tour. This approach performs two nearest neighbour heuristics at each tail, and the path with the shorter distance is selected. Reinelt (1994) uses the approach of insertion of forgotten cities to avoid adding too many isolated cities at the end. The approach is not to let the degree of the free city to fall below a certain pre-specified level (such as 2 or 3). The degree of the free city refers to the number of adjacent cities that has not been included in the current path. If a city falls below the pre-specified level, then it is added immediately to the path.

2.2.2 Insertion heuristics

There are three important considerations in insertion heuristics (Lawler et al, 1985): the choice of starting city, a selection criterion to select the most appropriate city to be inserted, and an insertion criterion which determines which part of the tour to insert the city. Examples of insertion heuristics include random, nearest and farthest insertions. These insertion heuristics varies in the way the city is inserted in the tour. For example, the nearest insertion heuristic starts with a tour of two cities that are nearest to one another. Then an unvisited city that is nearest to the tour cities is selected. This city is inserted between two consecutive cities that result in the minimum increase of tour length. The procedure of farthest insertion heuristic is similar to the nearest insertion with the exception that it selects two cities that are located farthest to one another as the initial tour. Then an unvisited city that is farthest to the tour is selected. In random insertion, a random city that results in the minimum increase in tour length is selected. For each heuristic, the selection and insertion steps are repeated until all

cities have been included in the tour. Table 2.1 shows the running time and results of worst-case analysis of the nearest, farthest and random insertion heuristics for an n-city problem. In general farthest insertion and random insertion heuristics outperform nearest insertion heuristic, because both farthest and arbitrary insertion heuristics produce good global tour (Reinelt, 1994).

Heuristics	Running time	Worst case (length of tour/length of optimal tour)
Nearest insertion	$O(n^2)$	≤ 2
Farthest insertion	$O(n^2 \log n)$	$\leq 2 \ln (n) + 0.16$
Random insertion	$O(n^2)$	$\leq 2 \ln (n) + 0.16$

Table 2.1: Running time and worst case behaviour of various insertion heuristics (Golden et al, 1980)

Gendreau et al (1992) integrates tour improvement within the insertion heuristic. The approach is called generalised insertion and unstringing and stringing procedure (GENIUS). The main feature of this approach is that insertion of a city does not necessary take place between two consecutive cities. However, the number of potential insertions is based on the neighbourhood of the city to be inserted to the tour. At the same time, an improvement is carried out in such a way that the improvement search is limited to the most promising moves. This way the reconnecting edge is able to join cities that are closest to one another. Their results show that the tour sometimes results in a shorter tour length compared to the standard insertion heuristics.

2.2.3 Christofides' heuristic

Christofides' heuristic uses minimal spanning tree as a basis to construct a tour. A spanning tree for a set of n cities is a collection of (n-1) edges that join all cities into a single connected tree (Johnson and Papadimitriou, 1985). Therefore a minimal

spanning tree is one with minimum cost. In this algorithm, the minimum length matching of the odd degree vertices are obtained from the minimal spanning tree. The tour is constructed by traversing the Euler cycle and selecting the cities in the order they are first encountered. However this approach only works if triangle inequality is satisfied. The running time is proportional to n^3, where n is the number of city, and the worst-case analysis shows that it is less than 1.5 times of the optimal tour (Reinelt, 1994).

2.2.4 Space filling curve heuristics

Space filling curve heuristic is one of the recent tour construction heuristic developed, which can be used to map city locations in Euclidean plane to a unit circle using the inverse of a closed space-filling curve (Bartholdi and Platzman, 1988). A tour is formed by visiting the cities in the order of their images appeared on the circle. Its running time is of the order O(nlogn), where n is the number of city, and worst-case analysis shows that it is 1.25 times the optimal tour for uniformly generated problems. This heuristic works well for non-uniformly generated problems and it performs particularly well with respect to the measure of the ratio of the longest link in the tour to the average link in the tour (Burke, 1994). The advantage of this approach is that it is fast and it can be used when the inter-city distance is unknown (Tate et al, 1994).

2.2.5 Neural network approach

Hopfield and Tank (1985) have shown that neural network can be used to solve TSP. This method works by dividing the network into a (n x n) two-dimensional array with 0-1 states. However, the original Hopfield and Tank's approach is only able to solve a 10-city problem. Yu and Jia (1993) applied a technique called the orthogonal array

table to overcome the shortcoming of Hopfield and Tank's approach. The use of orthogonal array table allows the most suitable parameters to be selected in search of better attracting regions that correspond to a better optimal solution (Yu and Jia, 1993). Problems with 10 and 30, 31, and 300 cities have been successfully solved using this approach. Angeniol et al (1988) implement another approach that is based on Kohonen's self-organising feature map (Kohonen, 1984). In this approach, a set of nodes is joined together dynamically in such a way that it can evolve continuously to claim each city in the tour. In another study, Burke (1994) uses a technique known as guilty net, which is based on Kohonen's self-organising feature maps, to solve non-uniformly generated problems to optimality.

2.3 Tour improvement heuristics

This section examines local search heuristic and other non-operational research global optimisation techniques that include simulated annealing, genetic algorithm and tabu search.

2.3.1 Local search heuristic

Local search heuristic is a tour improvement heuristic which systematically tries to improve a tour after an initial complete tour is found. Most commonly used local search heuristics include 2-opt, 3-opt and Lin-Kernighan heuristics (Lin, 1965; Lin and Kernighan, 1973). Theoretically r-opt heuristic (where r = 2, 3, 4, ...) improves the tour by deleting r existing edges and replacing with r new edges. For example, if r = 2, then two edges in the tour are deleted and two new edges are reconnected in a different way to obtain a new tour. If the exchange reduces the total distance of the tour, then

that tour becomes the current solution. If it does not, then another attempt to exchange two more edges are carried out. This process is repeated until a tour with the shortest distance is found. The running time for r-opt heuristic is proportional to n^r, where n is the number of city (Golden et al, 1980). Theoretically the larger the number of edges exchanged, the better the solution is. However, this means the number of exchanges needs to be carried out will also increase rapidly and computation cost is increased too. Thus 2-opt and 3-opt are usually performed (Golden and Stewart, 1985). In deciding which edge to be exchanged, it has been suggested that computational time can be saved if the edge with the longest distance can be exchanged at the first instance instead of selecting the edges at random (Christofides and Eilon, 1972).

Lin-Kernighan heuristic is regarded as the best improvement heuristic in the literature (Johnson, 1990). It is a variable depth r-opt in which the number of edges to be exchanged is decided dynamically at each iteration of improvement, and is not fixed (Lawler et al, 1985). Lin-Kernighan heuristic first employs breadth-first search and follows by depth-first search in deciding the number of edges of exchange at each iteration of improvement (Papadimitriou, 1992; Mak and Morton, 1993). It allows a tour length to increase during some stage of the improvement process if it opens up new possibilities for achieving considerable improvement later (Reinelt, 1994). For this reason it is able to find a better solution compared to r-opt heuristic.

A major weakness of local search heuristic is its tendency to get stuck at local optimum because the heuristic searches for improvement is within the local neighbourhood. The quality of the solution relies on local configuration. One way to improve the chance of finding better local optimum is to repeat the improvement process many times with different initial tours. In another study, Gu and Huang (1994)

apply the search space smoothing technique to avoid entrapment of local optimum. A pre-processing smoothing technique is applied to transform the objective function to a series of simplified functions. Then r-opt is applied to the simpler functions. The quality of the solution depends on how the original problem is reduced at each step, and how the intermediate solution is used to achieve global optimum. This technique is similar in concept to simulated annealing that will be described later.

Being trapped in local optimum is not the only drawback of local search heuristics. It has been proven that finding a local optimum using Lin-Kernighan heuristics for TSP is PLS-complete (polynomial-time local search) (Papadimitriou, 1992). In general, there is no guarantee to find a tour whose length is bounded by a constant multiple of optimal tour lengths, even if an exponential number of steps are allowed (Papadimitriou and Steiglitz, 1982). It is not known whether 2-opt and 3-opt are also PLS-complete, however it has been shown that for k>3, k-opt is also PLS-complete (Johnson, 1990).

2.3.2 Simulated annealing

The application of simulated annealing to solve optimisation problems was independently proposed by Kirkpatrick et al (1983) and Cerny (1985), and is based on the concept of the physical annealing process. The strength of simulated annealing lies in a process called uphill move. It allows a neighbourhood move that increases the value of the objective function with small and decreasing probability. The acceptance or rejection of an uphill move is determined using the Boltzmann probability function defined as Prob(E) = exp(-E/kT), where E is the change in energy, T is the temperature and k is the Boltzmann constant. The main feature of this approach is it allows the

system to jump out of local optimum by taking the uphill move that can lead to a new configuration or a new neighbourhood region so that a global optimum is found.

The quality of the solution depends on the annealing schedule, which determines the rate of cooling (Press et al, 1992; Koulamas et al, 1994; Lourenco, 1995). It is during this cooling process that simulated annealing sometimes accepts a higher cost function than the current solution. This constitutes the uphill move that ensures the solution is not trapped in local optimum (Lawler et al, 1985; Press et al, 1992; Koulamas et al, 1994; Lin and Hsueh, 1994; Jedrzejek and Cieplinski, 1995). Geometric cooling rule is usually used to determine the cooling rate (Lourenco, 1995). In general, experimentations are often required to determine how the temperature can be changed from higher to lower values as well as the amount of time it takes to reach equilibrium at that temperature.

The advantage of simulated annealing is it is not restricted to the problem domain, especially if the annealing schedule can be designed appropriately in such a way that the temperature is cooled slowly enough. Geman and Geman (1984) show that if the temperature is reduced slowly enough, then simulated annealing statistically guarantees to find an optimal solution for any arbitrary problem and the solution reaches to ground state with a logarithmic schedule. However in practice logarithmic cooling can be too slow to reach. Thus the most important factor is to be able to reduce the temperature very slowly. However, this will increase the running time, especially if the cost function is expensive to compute. In addition, for problems with a smooth energy landscape, the use of simulated annealing may be unnecessary as local search heuristics may be sufficient.

In general, simulated annealing is capable of providing fairly good solutions with short computational time if an appropriate annealing schedule is used. To demonstrate this feature, Usami and Kitaoka (1997) experimented with 9 known examples of 100 cities from literature, and optimal solutions were obtained for 3 problems within 1 minute of calculation using Pentium 90 MHz processor. On average the tour length is found to be only 0.5% longer than the optimal tour. Experiments conducted by Johnson (1990) show that simulated annealing can find a better tour than 3-opt and Lin-Kernighan heuristics if sufficient time is allowed. However the increase in running time is enormous.

Simulated annealing can be combined with other heuristics to improve its performance (Dowsland, 1995). Lin and Hsueh (1994) develop a hybrid method by combining nearest neighbour heuristic with low temperature simulated annealing. Their result is within 3 to 5 percent of the optimal value.

2.3.3 Genetic algorithm

Genetic algorithm is a global optimisation heuristic based on the principles of natural selection and population evolution (Holland, 1975). The principle is to identify high quality properties that can be combined into a new population so that the new generation of solutions are better than the previous population (Kolen and Pesch, 1994). Unlike other heuristics that consider only one solution, genetic algorithm considers a population of feasible solutions. The algorithm consists of four components: selection, crossover, mutation and replacement. The algorithm can be described as follow.

The population are initialised by randomly generated feasible solutions. Each feasible solution is assigned a fitness value. This value is used to determine the probability of choosing a solution as the parent solution. An example of fitness value is the tour length, and the probability of selecting it as a solution is inversely proportional to the length. The solutions with high fitness values will be selected to breed with other parent solutions in the crossover step, which can be carried out by exchanging part of the tour with another. The aim is to combine good characteristics of parent solutions to create new children solutions. It is important in this step to determine an appropriate crossover point, such as which edges should be selected so that they can be passed to the children solutions. Solutions are mutated through the changes made to the children solutions. The aim of mutation step is to ensure diversity in the population. It is not necessary to perform mutation step to every solution. A portion of the solution can have one or more edges exchanged using some assigned probability. The last step is the replacement of current population in which the parent generation is replaced with the new population of children solutions. The process is repeated until a convergence criterion is satisfied. This can be achieved by repeating for a specific number of generations or until the population does not show any further improvement (Laporte et al, 1996).

Crossover and mutation are two important steps in genetic algorithm. Crossover ensures better children solutions are generated in the new generation, and mutation ensures uphill move is allowed. These two steps form the strength of genetic algorithm. More importantly, genetic algorithm conducts the search of optimal solution based on the population; in contrast to a single feasible solution in other optimisation techniques. This allows genetic algorithm to take advantage of the fittest solution by assigning higher probability that can result in better solution. However, it is necessary

to use an appropriate fitness function so that the constraints of only one city can be visited at one time is taken into consideration.

It is important to find good parameter settings in order for genetic algorithm to work. One of the factors is the determination of population size, because if the population is too small, a premature convergence may occur which leads to local optimum. On the other hand, if the population is too large, then there may be a significant increase in computation time because too many solutions need to be considered. Other factors include determination of the crossover point in parent solutions and strategies for mutation. The construction of crossover operators should not result in children solutions that are too far away from the parent solutions. Similarly, if too many edges are selected during the mutation step, it will also increase computation time as too high mutation may result in too much diversity. On the other hand, too low mutation may result in a sub-optimal solution. Evaluation of fitness values is also important, because a too simplistic fitness function may lead to convergence of local optimum.

Generally genetic algorithm is used as a meta-heuristic that incorporates other improvement heuristics such as Lin-Kernighan heuristic. Kolen and Pesch (1994) included local search (2-opt and Lin-Kernighan heuristics) with genetic algorithm. Their results show that for large problem size, this technique performs better especially if there is severe time constraint imposed on the running time. In another study, Tate et al (1994) use genetic algorithm to improve tours that have been generated by space filling curves. Yip and Pao (1995) develop a technique called guided evolutionary simulated annealing that combines simulated annealing with simulated evolution. There are two levels of competition: competition-within and competition-between the families. Competition within the family is based on simulated annealing, and

competition between families measures the fitness of each family that determines the number of children that should be generated in the next generation. Experiments have been conducted using 10-city and 50-city problems, and results show that there is a 100% convergence in all test problems.

2.3.4 Tabu search

In order to escape local optimum, a global optimisation technique such as simulated annealing allows tour length to increase during the process. However, no step is taken to prevent the heuristic from revisiting the same local optimum again. Tabu search is designed to overcome this problem (Glover, 1990). Tabu search employs short- and long-term memory strategies. Short-term memory consists of a tabu list that is used to determine if a move is allowed. Long-term memory is used to escape from the local optimum and redirect the search to other neighbourhoods. Knox (1994) calls this strategy a supervisory heuristic that guides the lower level heuristic performing the actual manipulations.

Tabu restriction and aspiration are two important features in this approach. When evaluating the neighbourhood of a solution, some potential solutions are classified as tabu or inadmissible. This results in tabu restriction that forbids the move. A tabu move can become admissible when an aspiration criterion is satisfied, and it can override the tabu move. An example of aspiration criterion is when a tabu move can produce a tour that is better than the current best tour (Glover, 1990).

Tabu list establishes a basis for deciding whether a move under consideration is forbidden or otherwise. For example, if 2-edge exchange is used, then a tabu list stores the edges that have been deleted. This way any future move that tries to introduce

those two edges to the tour is forbidden. Tabu search alternates between different types of neighbourhoods (Laporte et al, 1996). It does not stop at the first local optimum. The search only stops when a predetermined number of iterations or processing times have elapsed. When a local optimum is reached, the search will select a bad move that has not been previously examined (Knox, 1994). The principle is that the best move that is not tabu is selected, even though it may result in an increase in cost. This way the search alternates between different neighbourhoods and allows it to escape from poor local optimum and move to other local optimum nearby, resulting in reaching new neighbourhood (Laporte et al, 1996). There are different strategies to identify the size of the tabu list, Knox (1994) recommends the length of the tabu list be kept at 3n, where n is the number of cities. Another approach is to use frequency-based information such as the frequency with which a move occurs in the search. For example, in an attempt to diversify the search, frequently occurring moves can be classified as tabu (Glover and Laguna, 1993; Xu and Kelly, 1996).

Results from Knox (1994) suggest that tabu search outperforms 2-opt and 3-opt when the size of the problems increases. The advantages of tabu search include independence of problem domain and flexible memory structure. However the success of the technique is dependent on selecting appropriate parameters for the tabu list.

Table 2.2 summarises the features, advantages and disadvantages of tour improvement heuristics that have been discussed above.

Method	Feature	Advantages	Disadvantages
r-opt	The use of neighbourhood structure in search of optimal solution	Search within it its local neighbourhood	Get stuck at a local optimum configuration
Simulated annealing	The use of annealing schedule to search for global optimal solution	Easy to implement Provide reasonable solutions	Long running time needed for convergence
Genetic algorithm	The use of crossover and mutation in a population of feasible solutions	Domain independence Robust Ease of modification Parallel nature	Difficult to implement crossover operation to ensure that the problem structure is reflected during the crossover process
Tabu search	The use of tabu restriction and aspiration criteria to forbid and override a move respectively	Independent of problem domain Flexible memory structure	Solution quality depends on appropriate management of tabu restriction and aspiration criteria

Table 2.2: Summary of local and global optimisation techniques

2.4 Branch and bound method

Branch and bound is an exact method in Operation Research that can be used to find optimal solution. In general, the branch and bound method solves problems by breaking up feasible solutions into a collection of subproblems. The approach performs branching and bounding operations and testing of elimination rule (Gendron and Crainic, 1994). The branching process partitions the problem into subproblems, and the bounding process keeps track of the best candidate found so far based on the upper and lower bounds of the subproblem (Baker, 1974). The search space can be represented in a form of a tree, where the root represents the original problem and the children of a given node as the subproblems obtained by the branching process (Viswanathan and Bagchi, 1993). The subproblems generated from the branching process are mutually exclusive. While the tree is generated, each child of a given node is said to be in one of three states: generated, evaluated or examined (Gendron and Crainic, 1994). A generated subproblem is evaluated when a bounding process has

been applied. It is examined to determine whether a branching operation needs to be applied to it or whether the elimination rule is applied to show that it can be pruned off. Subproblems that do not lie within the bound are eliminated from further consideration. In each instance, lower and upper bounds are revised. The process is repeated until the optimal solution is found. A common approach to generate subproblems is to use the Carpaneto and Toth's branching rule (Carpaneto and Toth, 1980). This branching rule generates subproblems by including and excluding certain edges during the branching process. In the branch and bound approach, the order of the tree is often depth-first search (Viswanathan and Bagchi, 1993). The disadvantage of this is possible erroneous decisions made cannot be corrected until late in the search process (Lawler et al, 1985).

The performance of branch and bound algorithm depends on the quality of the lower and upper bounds. Experiments have shown that assignment problem can provide an excellent lower bound for asymmetric problem (Lawler et al, 1985; Miller and Pekny, 1991). For symmetric problem, 1-tree is commonly used as lower bound (Reinelt, 1994). The upper bound can be set at an arbitrary large value. Alternatively, the upper bound can be determined using heuristics. For example, Karp's patching algorithm (Karp, 1979), which is based on assignment problems with a patching operation that joins subtours into one by deleting and inserting edges in the subtours, is commonly used as upper bound (Miller and Pekny, 1991; Carpaneto et al, 1995; Zhang, 1999).

Padberg and Rinaldi (1991) develop branch and cut algorithm to solve large symmetric TSP. The lower bound in this approach is obtained from linear programming relaxations. According to Padberg and Rinaldi (1991, p.62), the core of the algorithm is "the polyhedral cutting plane procedure that exploits a subset of the system of linear

inequalities defining the convex hull of the incidence vectors of the Hamiltonian cycles of a complete graph". There are four key elements in this algorithm: a heuristic procedure to find good upper bound, the quality and quantity of cuts generated, efficient use of linear programming solver, and an efficient tree search approach that combines branching with cutting plane.

2.5 Conclusion

This chapter has reviewed various commonly used heuristics to solve traveling salesman problem. From literature review, it can be seen that extensive research has focussed on designing and developing methods that are capable of improving tour so that an optimal solution is obtained. However, there has been relatively little research on using tour construction heuristics to obtain optimal solution. In fact, tour that is built using tour construction heuristics often needs to be improved using tour improvement heuristics so that a better solution can be found. The poor quality of solution obtained using tour construction heuristics can be attributed to these algorithms being greedy approaches. The tour configuration does not change during the tour construction process and the tour is often constructed using local distance information, which results in the approach being myopic. On the other hand, the key element of local search heuristics is to use edge-exchange method to change the tour configuration in such a way that other neighbourhood can be explored. This way a better solution can be found. Similarly in global optimisation heuristics, the aim is to ensure that other neighbourhood regions can be explored so that the solution is not locally optimal.

Literature review has also shown that traveling salesman problem can be solved to optimality using operations research method such as branch and bound. At the same time, researchers have used concepts in non-operations research area such as physical science (simulated annealing), biology (genetic algorithm) and artificial intelligence (neural networks) to solve the problem. However, there is little research in using heuristic learning algorithm to address the problem. In particular, using heuristic learning algorithm to investigate methods that allow tour configuration to change during the tour construction process and to address the greedy and myopic nature of tour construction heuristics. This research seeks to address these issues by demonstrating that the well-developed heuristic learning algorithm in artificial intelligence may present another approach in addressing the traveling salesman problem. In particular, the research aims to investigate the application of a heuristic learning algorithm to address the greedy and myopic natures of tour construction heuristics.

CHAPTER 3: TOUR CONSTRUCTION USING HEURISTIC LEARNING APPROACH

3.1 Introduction

In traditional tour construction heuristics, the tour is built from scratch and the node is added one at a time until a complete tour is found. This is a greedy approach in which part of the tour that is already built remains unchanged, and no attempt is made to change the tour configuration. In addition, this approach is myopic as it often relies on local distance information to build tour. Reinelt (1994) points out that in general, constructing a tour using only tour construction heuristics alone will not lead to optimal solution. It often needs to be improved using tour improvement heuristics such as 2-opt or Lin-Kernighan heuristics, which made improvement to the tour by exchanging edges until a shorter tour is found. This chapter investigates the application of a heuristic learning algorithm to address the greedy and myopic natures of tour construction heuristics.

A heuristic learning algorithm called Search and Learning A* algorithm (SLA*) will be applied. The feature of SLA* is the application of heuristic learning to update and improve the initially under-estimated heuristic estimates, which will lead to the improvement of state selection decisions and an optimal solution when the goal state is reached. Before SLA* can be applied, the problem needs to be transformed to a state-space representation. Therefore a state-space transformation process, which consists of state definition, transition cost and a state transition operator will be investigated. In

addition, the implementation of SLA* requires a non-overestimating heuristic estimate to be determined. The heuristic estimate of a state represents an estimate of solution from that state to the goal state, therefore a suitable method to compute the heuristic estimation is important and will be investigated. The heuristic learning mechanism of SLA* allows the algorithm to update the heuristic estimates of visited states, and thus modify the tour configuration along the search process. However, to prevent the search space from becoming too large when SLA* is implemented, a suitable search strategy based on the geometric properties of TSP will be explored.

This chapter is organised as follows. Section 2 examines the Search and Learning A* algorithm (SLA*). Section 3 investigates implementation issues when SLA* is applied. The investigation includes identifying the state-space transformation process, and an appropriate method of computing non-overestimating heuristic estimate. A search strategy that utilises the concept of Delaunay triangulation will be examined. Section 4 outlines the step-by-step application procedure of SLA*-TSP algorithm (Search and Learning A* algorithm for Traveling Salesman Problems). An example is included to demonstrate the working of the algorithm. Section 5 examines the approach of finding approximate solutions by introducing the notion of learning threshold to SLA*-TSP. The conclusion follows in Section 6.

3.2 Search and Learning A* algorithm

Search and Learning A* algorithm (SLA*) is a real time admissible heuristic learning algorithm, which uses the heuristic evaluation function to estimate the relative merit of different states to the goal state (Zamani, 1995). The search path improvement feature

of SLA* requires the initial estimate of a state to be non-overestimating so that it may be improved during the search process. The algorithm of SLA* is described as follows.

The rationality of this algorithm is that a state that is further away from the goal state should have a larger heuristic estimate. From a front state x, the state selection process is based on the minimum increment of the heuristic function $f(y) = k(x,y) + h(y)$, where $k(x,y)$ is the positive true edge cost from state x to its neighbouring state y, and $h(y)$ is the heuristic estimate of state y. The algorithm will first identify the state with the minimum $f(y)$, and then compares it with $h(x)$ to decide if $h(x)$ can be improved. If it does, heuristic learning is said to occur. This relationship can be expressed as $h(x) \geq \min \{k(x,y) + h(y)\}$. Hence, if $h(x)$ is not smaller than minimum $f(y)$, then this relationship is true, and the state with minimum $f(y)$ is added to the search path as the new front state. From this new state, the algorithm continues the search operation. If this heuristic relationship is not true, then $h(x)$ is too much under-estimated and it can be updated to this minimum $f(y)$, then the new $h(x)$ is replaced with the minimum of $\{k(x,y) + h(y)\}$, and still remains as non-overestimating.

Due to the fact that the selection criterion of a state is based on the heuristic estimate of its neighbouring states, the update of $h(x)$, a larger value than before, may invalidate the selection of its previous state (x-1). In order to reflect the effect of the new $h(x)$ to the search path, the algorithm conducts a backtracking operation by applying the above rationale to the state (x-1) to see if $h(x-1)$ can be updated. If $h(x-1)$ is updated, then its previous state (x-2) will need to be reviewed too. In this way, the states of the search path will be reviewed one by one in the reverse order. Along the way, any state whose heuristic estimate has been improved is detached from the path, because the improvement casts doubt on the validity of its previous minimum heuristic function

status. This review process continues until it reaches either the state whose heuristic estimate remains unchanged after it has been examined for heuristic learning, or the root state if the algorithm backtracks all the way to the root state, and then the algorithm resumes the search from this state. As a result, before the resumption of the search path, the algorithm would have completely updated the earlier path. Hence the search path that is to be developed subsequently, before the next heuristic learning, will be a minimum path. When the goal is reached, the path is an optimal path and represents a complete solution. This algorithm can be expressed in the following steps:

Let $k(x,y)$ be the positive edge cost from state x to a neighbouring state y, and

$h(x)$ be the non-overestimating heuristic estimate from state x to the goal state.

Step 1: Put the root state on the backtrack list called OPEN.

Step 2: Call the top-most state on the OPEN list x. If x is the goal state, stop; otherwise continue.

Step 3: Evaluate $[k(x,y) + h(y)]$ for every neighbouring state y of x, and find the state with the minimum value. Call this state x'. Break ties randomly.

Step 4: If $h(x) \geq [k(x,x') + h(x')]$, add x' to the OPEN list as the top-most state and go to step 2. Otherwise replace $h(x)$ with $[k(x,x') + h(x')]$.

Step 5: If x is not the root state, remove x from the OPEN list.

Step 6: Go to step 2.

3.3 Implementation issues

This section investigates the implementation issues of SLA* on Euclidean TSP, where cities are given as points in a two-dimensional plane and their distance is computed using Euclidean distance. In order to apply SLA*, a state-space transformation method that can formulate the problem into state-space problem must first be developed. A state-space transformation approach, which consists of state definition, state transition operator and state transition cost, will be examined. Next, a suitable method to compute the heuristic estimation will be investigated. Finally, a search strategy that utilises geometric properties of TSP is explored. The concepts of Delaunay triangulation will be examined and explored so that an efficient search approach can be developed. This is important to ensure that the search space does not become too large, which can influence the performance of the algorithm.

3.3.1 State space transformation approach

The state space transformation process includes state definition, state transition operator and state transition cost.

Definition of state

A state is defined as a tour, consisting of selected cities and the remaining unvisited cities of a given problem. The selected cities form a partially incomplete tour, which is a closed tour by connecting cities in the order of their selection and connecting the last city to the city of origin. A partial tour becomes a complete tour when all cities are included.

The status of a city at a given moment is in one of the following two sets: *partial tour* P, or *unvisited cities* U. The root state consists of the city of origin, which can be selected randomly. Let the city of origin be denoted as city 1, then the root state can be expressed as {(1,1),U}, with U=(2,3,...,n), and the goal state is {(1,2,...,n,1), ∅}. A given state S_i, can be represented as ((1,2,…,i,1),{U}), where i is the last city added to the tour and U is (i+1,i+2,...,n).

State transition

For a given state S_i, {(1,2,…,i,1),(U)}, where i is the last city added to the partial tour, the transition to the next state S_{i+1} is through the selection of a city from the neighbouring cities of i, which are in the unvisited set U. The selection criterion is the minimum increment of tour length.

State transition cost

The transition cost from a parent state S_i to a child state S_{i+1} is the increment in distance between states S_i and S_{i+1}, which is [d(i,i+1)+d(i+1,1)-d(i,1)] with d(i,j) = $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ being the Euclidean distance between two cities $i(x_i,y_i)$ and $j(x_j,y_j)$.

3.3.2 Heuristic estimation of each state

A minimal spanning tree is a spanning tree that connects all nodes such that its cost is minimum (Lawler et al, 1985). Among the possible lower bound estimates of a partial tour, minimal spanning tree is used to compute the non-overestimating heuristic estimate. For a state S_i, the heuristic estimate is the cost of the minimal spanning tree on the remaining (n-i) unvisited cities, where n is the number of cities. Either Prim's

algorithm (Prim, 1957) or Kruskal's algorithm (Kruskal, 1956) can be used to calculate the minimal spanning tree. In this research, Kruskal's algorithm is used. It works by maintaining a set of partial minimum spanning trees, and repeatedly adds the shortest edge so that it joins two trees together. If a cycle is formed, then this edge is ignored and the next shorter edge is added. The process is repeated until (n-1) edges are added to form the spanning tree, where n is the number of nodes. The running time of Kruskal's algorithm is O((n+m)log(n+m)), where m and n are the number of edges and nodes respectively (Mehlhorn and Näher, 1999).

3.3.3 Search strategy

The size of the solution space of the traveling salesman problem is exponential in term of problem size, therefore it is necessary to control the search space so that it is not rapidly becoming too large. An efficient search strategy is very important. Otherwise, the number of possible tours to be considered will increase rapidly. The approach is to consider only those edges that are likely to result in an optimal tour, and useless edges that are unlikely to result in optimal tour should not be considered in the search process. The geometric properties of the point sets will be exploited for this purpose. One of the computational geometry concepts that can be used to obtain information about the structure of point sets is Delaunay triangulation. Therefore, a searching framework that is based on properties of Delaunay triangulation will be examined. In the following, the concept of Delaunay triangulation will be explained. However, the concept of Voronoi diagram, which is a geometric dual of Delaunay triangulation, will be examined first.

The concept of Voronoi diagram is based on proximity of points in the plane. It partitions the plane into a set of polygons, called Voronoi regions, so that each polygon

consists of points closer to one than to any others. Reinelt (1994) defines Voronoi diagram as follows: the Voronoi diagram divides the set of points into a set of polygons of which the boundaries are perpendicular bisectors between two points. The Voronoi diagrams can be constructed using the divide and conquer algorithm, in (nlogn) time (O'Rourke, 1998). The algorithm is given in Figure 3.1.

Step 1:	Partition S into two subsets S_1 and S_2 of approximate size. (This step can be done using median value of x-coordinates.)
Step 2:	Construct the Voronoi diagram of S_1 and S_2 recursively.
Step 3:	Merge Voronoi diagram of S_1 and S_2 to form the total diagram.

Figure 3.1: Divide and conquer algorithm (O'Rourke, 1998)

Voronoi diagram can be used to solve the nearest neighbour problem, and it allows the proximity question to be answered. Examples of applications that can be solved using Voronoi diagram include cluster analysis, collision detection, facility location, path planning, associative file sharing and others (Preparata and Shamos, 1985; Aurenhammer, 1991; O'Rourke, 1998). In cluster analysis, Voronoi diagram is used to partition a set of data into groups in which similar data are organised. In facility location problem, Voronoi diagram is used to identify a site in which a facility, such as a shop, can be built farthest from its nearest competitor. In collision detection, Voronoi diagram is used for proximity detection so that a robot can stop before a collision occurs.

Delaunay triangulation is the geometric dual of Voronoi diagram. Therefore, the structure of Delaunay triangulation is very closely related to Voronoi diagram. The duality of Voronoi diagram and Delaunay triangulation implies that the edges of

Delaunay triangulation are orthogonal to their corresponding Voronoi edges (Reinelt, 1994). It can be constructed based on the largest empty circle property derived from Voronoi diagram, which states no other point in the point sets should fall in the interior of the circumcircle of any triangle in the triangulation (Aurenhammer, 1991; O'Rourke, 1998). The running time of this approach is of the order O(nlogn), where n is the number of points (O'Rourke, 1998). The procedure of the largest empty circle is shown in Figure 3.2.

Step 1:	Compute the Voronoi diagram $VR(P_i)$ for all i = 1,2,...,n.
Step 2:	Compute the convex hull H.
Step 3:	For each Voronoi vertex v do
	if v is inside H then compute radius of circle centred on v and update maximum value.
Step 4:	For each Voronoi edge e do
	compute the intersection of e with the convex hull boundary, call this point p,
	compute radius of circle centred on p and update maximum value,
Step 5:	Return maximum value.

Figure 3.2: Largest empty circle algorithm (O'Rourke, 1998)

There are two important properties of Delaunay triangulations that are related to this research: the boundary of Delaunay triangulation is the convex hull of the point sets, and minimal spanning tree is a subset Delaunay triangulation (Aurenhammer, 1991; O'Rourke, 1998). For an n-city problem, there are at most (3n-6) edges and (2n-4) triangles formed by Delaunay triangulation (Reinelt, 1994). Although in general Delaunay triangulation does not contain a traveling salesman tour, it has been shown

that there is a high probability that the edges appear in the optimal tour of TSP are also edges of Delaunay triangulation (Aurenhammer, 1991; Stewart, 1992; Krasnogor et al, 1995; Phan, 2000). Therefore, Delaunay triangulation can provide good information about the location of promising edges to be considered as part of the optimal tour. It is obvious that one can utilise Delaunay triangulation as a search strategy to locate promising neighbouring city that will lead to optimal tour. Figures 3.3 and 3.4 show Voronoi diagram and Delaunay triangulation of an 8-city problem respectively.

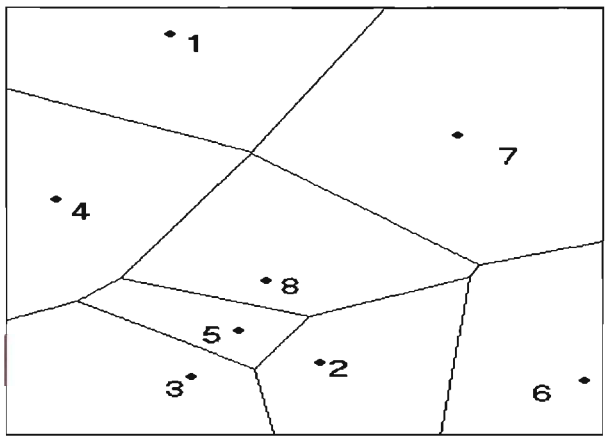

Figure 3.3: Voronoi diagram of an 8-city problem

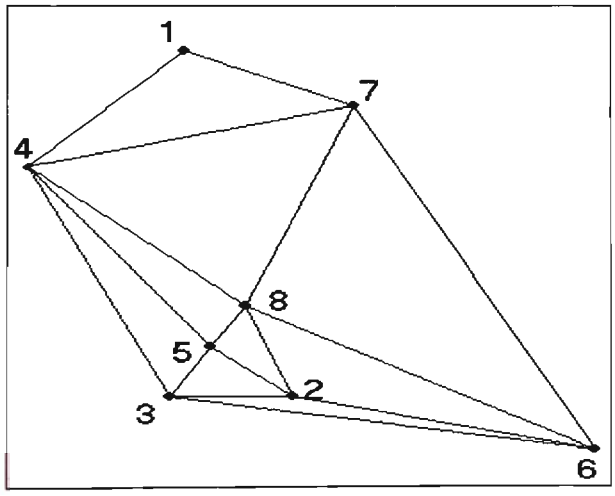

Figure 3.4: Delaunay triangulation of an 8-city problem

3.4 The proposed heuristic learning algorithm – SLA*-TSP

This section describes SLA* algorithm that takes account of Delaunay triangulation as part of the search strategy. This algorithm was developed using the state space transformation process and the heuristic estimation approach using minimal spanning tree identified in the previous section. In the algorithm, the completed tour length for state S_i is computed rather than the heuristic estimate of S_i. This allows the estimated tour length to be compared. However, the result is the same as that when the heuristic estimate is computed. The algorithm, acronym as SLA*-TSP (Search and Learning Algorithm for Traveling Salesman problems), is given as follows:

Let S_i be the i^{th} state with its tour $P_i(1,2,...,i,1)$, where 1 is the city of origin and i is the last city of the tour. Its heuristic estimate h(i) is the minimum spanning tree of the remaining (n-i) cities. S_i is the goal state when i = n.

d(i,j) be the Euclidean distance between city i and city j.

H(i) be the estimated tour length for S_i, which consists of the tour p_i and h(i).

Step 0: Apply Delaunay triangulation algorithm to find neighbouring nodes for each city.

Step 1: Locate the city of origin as the one with the smallest x-coordinate; choose the city with the largest y-coordinate to break ties.

Step 2: Put the root state on the backtrack list called OPEN.

Step 3: Call the top-most state on the OPEN list S_i. If S_i is the goal state, stop. Otherwise continue.

Step 4: Find the $(i+1)^{th}$ city with min{[d(1,2)+d(2,3)+...+d(i-1,i)+d(i,i+1)+d(i+1,1)] +h(i+1)} from neighbouring cities of i; break ties randomly. If no neighbouring city of i can be found, go to step 6.

Step 5: If {[d(1,2)+d(2,3)+...+d(i-1,i)+d(i,1)] + h(i)} ≥ min{[d(1,2)+d(2,3)+...+d(i-1,i)+d(i,i+1)+d(i+1,1)] + h(i+1)}, add S_{i+1} to the OPEN list as the top-most state; otherwise replace h(i) with [d(i,i+1)+d(i+1,1) + h(i+1) - d(i,1)].

Step 6: Remove S_i from OPEN list if it is not the root state.

Step 7: Go to step 3.

3.4.1 Example

An example of an 8-city problem (see Figure 3.4, page 35) is included in this section to demonstrate the working of SLA*-TSP. This section describes the procedures when the algorithm is applied. The problem is obtained from Gendreau et al (1992). The complete search process in finding an optimal solution is summarized in Table 3.1. In the table, each row represents one attempt to locate a tour as far as possible without incurring heuristic learning. The search path in each row is disrupted as soon as the estimated tour length of a newly selected state is greater than that of its parent state. Thus the last state of each row is the one that has a larger estimated tour length and causes the backtracking. When this happens, the algorithm will update the estimated tour length of its parent state. The upper entry in each cell represents a state, and the lower entry is the estimated tour length of that state. The search process consists of

both forward searching and backward updating. In order to maintain the simplicity of the table, only the forward searching part in each row is presented. The heuristic updating and the backtracking process of the algorithm is not shown, instead an asterisk ("*") is used to indicate the state where backtracking ends and the new round of forward search begins. The following describes the steps and procedures when SLA*-TSP is applied.

Iteration	Root state	Level-1	Level-2	Level-3	Level-4	Level-5	Level-6	Level-7
1	(4,4) * 13070	(4,1,4) 15891						
2	(4,4)* 15891	(4,1,4) 15891	(4,1,7,4) 15740	(4,1,7,8,4) 18548				
3	(4,4)* 18548	(4,1,4) 18548	(4,1,7,4) 18548	(4,1,7,8,4) 18548	(4,1,7,8,5,4) 18464	(4,1,7,8,5,3,4) 18409	(4,1,7,8,5,3,2,4) 16964	(4,1,7,8,5,3,2,6,4) 24122
4	(4,4)* 19348	(4,7,4) 19348	(4,7,8,4) 22569					
5	(4,4)* 20415	(4,5,4) 20415	(4,5,3,4) 20360	(4,5,3,2,4) 22171				
6	(4,4)* 20424	(4,8,4) 20424	(4,8,5,4) 21037					
7	(4,4)* 20740	(4,3,4) 20740	(4,3,5,4) 20360	(4,3,5,8,4) 20982				
8	(4,4)* 20982	(4,3,4) 20982	(4,3,5,4) 20982	(4,3,5,8,4) 20982	(4,3,5,8,2,4) 21650			
9	(4,4)* 21037	(4,5,4) 21037	(4,5,8,4) 21037	(4,5,8,7,4) 23310				
10	(4,4) 21037	(4,8,4)* 21037	(4,8,5,4) 21037	(4,8,5,3,4) 20982	(4,8,5,3,2,4) 21054			
11	(4,4)* 21054	(4,8,4) 21054	(4,8,5,4) 21054	(4,8,5,3,4) 21054	(4,8,5,3,2,4) 21054	(4,8,5,3,2,6,4) 21678		
12	(4,4)* 21429	(4,1,4) 21429	(4,1,7,4) 21429	(4,1,7,8,4) 21429	(4,1,7,8,2,4) 21429	(4,1,7,8,2,5,4) 20744	(4,1,7,8,2,5,3,4) 16793	(4,1,7,8,2,5,3,6,4) 26402
13	(4,4) 21429	(4,1,4) 21429	(4,1,7,4) 21429	(4,1,7,8,4) 21429	(4,1,7,8,2,4)* 21429	(4,1,7,8,2,3,4)* 21142	(4,1,7,8,2,3,5,4) 16263	(4,1,7,8,2,3,5,6,4) 26236
14	(4,4)* 21445	(4,1,4) 21445	(4,1,7,4) 21445	(4,1,7,8,4) 21445	(4,1,7,8,5,4) 21445	(4,1,7,8,5,2,4) 21445	(4,1,7,8,5,2,3,4) 16643	(4,1,7,8,5,2,3,6,4) 26252
15	(4,4)* 21650	(4,3,4) 21650	(4,3,5,4) 21650	(4,3,5,8,4) 21650	(4,3,5,8,2,4) 21650	(4,3,5,8,2,6,4) 22274		
16	(4,4)* 21678	(4,8,4) 21678	(4,8,5,4) 21678	(4,8,5,3,4) 21678	(4,8,5,3,2,4) 21678	(4,8,5,3,2,6,4) 21678	(4,8,5,3,2,6,7,4) 21210	(4,8,5,3,2,6,7,1,4) 22300
17	(4,4)* 22171	(4,5,4) 22171	(4,5,3,4) 22171	(4,5,3,2,4) 22171	(4,5,3,2,8,4) 20353	(4,5,3,2,8,6,4) 23384		
18	(4,4)* 22274	(4,3,4) 22274	(4,3,5,4) 22274	(4,3,5,8,4) 22274	(4,3,5,8,2,4) 22274	(4,3,5,8,2,6,4) 22274	(4,3,5,8,2,6,7,4) 21806	(4,3,5,8,2,6,7,1,4) 28896
19	(4,4)* 22300	(4,8,4) 22300	(4,8,5,4) 22300	(4,8,5,3,4) 22300	(4,8,5,3,2,4) 22300	(4,8,5,3,2,6,4) 22300	(4,8,5,3,2,6,7,4) 22300	(4,8,5,3,2,6,7,1,4) 22300

Table 3.1: Summarised search process for 8-city problem

Initially the neighbouring edges for each city in the problem are identified using Delaunay triangulations (see Figure 3.4, page 35). In this example, the city of origin is selected using the minimum x-coordinate. Therefore the city of origin is city 4: the corresponding root state is {(4,4),U}, and its heuristic estimate to the goal state is the minimal spanning tree of the remaining unvisited cities U (i.e. cities 1,2,3,5,6,7,8), which is 13070. For simplicity, from hereafter only the tour part of a state is shown, and its unvisited states U is not given. The selection of the next city to join the tour is made from the five neighbouring cities (1,3,5,7,8) of city 4 obtained from the edges of Delaunay triangulations. Thus the corresponding front states are: (4,1,4), (4,3,4), (4,5,4), (4,7,4) and (4,8,4). Step 4 of the algorithm makes the selection by finding the state with the smallest estimated tour length. As shown in row 1, state (4,1,4) is selected with a smaller value of 15891 = 10785 (heuristic) + 5106 (tour), which is greater than 13070 of its parent state (4,4). Step 5 of the algorithm updates the parent state with this new value 15891, and backtracks to its parent state (4,4).

In row 2, the algorithm starts from (4,4). It again selects (4,1,4), and because of no heuristic updating involved, the state (4,1,4) becomes the new front state. For the state selection of the next round, the candidate edge set shows that promising neighbouring cities include 1, 4 and 7, as shown in Figure 3.4. Since the last city that was added to the tour is 1, then the states to be considered are those of its neighbouring cities 4 and 7. City 4 is not eligible because it was in the tour already. Hence, the next state to consider is (4,1,7,4). Its estimated tour length is 15740 = 7154 (heuristic) + 8586 (tour). This value is not greater than 15891 of its parent state, hence state (4,1,7,4) becomes the new front state. From city 7 (which was the last city added to the partial tour), only cities 6 and 8 are eligible for consideration. The estimated tour lengths of these two corresponding states are 23093 and 18548 respectively. State (4,1,7,8,4) is

selected for having a smaller value of 18548. However, this value is larger than 15740 of its parent state (4,1,7,4). Hence, the estimated tour length of (4,1,7,4) is updated to 18548, and the algorithm backtracks to (4,1,4). From (4,1,4), the state (4,1,7,4) is selected with an estimated tour length of 18548, which updates that of (4,1,4) to 18548. The algorithm then backtracks to the root state (4,4). The selection of state (4,1,4) leads to the update of the estimated tour length of state (4,4) from 15891 to 18548.

Row 3 shows the search from the newly updated state (4,4). The search path shows states (4,1,4), (4,1,7,4), (4,1,7,8,4), (4,1,7,8,5,4), (4,1,7,8,5,3,4), (4,1,7,8,5,3,2,4) and stops at (4,1,7,8,5,3,2,6,4). At state (4,1,7,8,5,3,2,6,4), its estimated tour length 24122 is greater than 16964 of its parent state (4,1,7,8,5,3,2,4). This causes the algorithm to backtrack. The algorithm backtracks all the way to the root state (4,4), and its estimated tour length is updated to 19348. With this new value, row 4 shows the next search path.

In row 4, there are three possible child states: (4,7,1,4), (4,7,6,4) and (4,7,8,4). The state (4,7,1,4) is pruned from consideration because city 1 does not have an unvisited neighbouring city to allow the state generation to continue. Thus, there are only two possible child states: (4,7,6,4) and (4,7,8,4). It is found that state (4,7,8,4) with tour estimated tour length of 22569 is the state with the minimum estimated tour length. This value is greater than 19348 of its parent state, and causes the algorithm to backtrack. The algorithm backtracks to the root state (4,4), and its estimated tour length is updated to 19348. With this new value, row 5 shows the next search path.

The same procedure is followed and steps are repeated until there is no more backtracking and heuristic learning, in which case optimal solution is found. Table 3.1

shows the summarised search process. The table shows that the algorithm takes 19 forward search trials to reach an optimal solution of 22300 with the optimal tour being (4,8,5,3,2,6,7,1,4).

3.5 SLA*-TSP with learning threshold

For problems where optimal solutions with reasonable computation time and cost are not feasible, the next best approach is to find approximate solutions of known quality. Search and learning A* algorithm with learning threshold is an algorithm that produces solutions guaranteed to be within a specific range of optimal solution (Zamani, 1995). SLA* with learning threshold can be applied to TSP with the knowledge that one can predict the quality of the approximate solution, because the solution found is within the range of this learning threshold. The algorithm works by employing learning threshold as an agent that delays the backtracking operations. The rationale of SLA*-TSP with learning threshold is explained as follows.

Before the algorithm with learning threshold is applied, one needs to decide the quality of the desired approximate solution. This is measured by how far the approximate solution should be positioned away from the optimal solution. This range will be the value of the learning threshold to be included in SLA*-TSP. The concept of learning threshold is used to delay the backtracking operations. In SLA*-TSP, backtracking takes place as soon as the minimum value of heuristic estimate of the front state exceeds its parent state. When this occurs, heuristic learning is said to have taken place. In the case of SLA*-TSP with learning threshold, backtracking may not necessary take place when heuristic learning occurs. Instead, it will only take place when the accumulated heuristic learning exceeds the learning threshold. Therefore

every time a heuristic learning takes place, the learning threshold is reduced by the amount of heuristic learning incurred. In this case, the value of the learning threshold is constantly being updated, and backtracking only takes place when the heuristic learning exceeds the updated value of learning threshold. Then backtracking operation occurs, and it will stop either until the root state is reached or until the heuristic learning no longer exceed the updated value of learning threshold. When the accumulated heuristic learning does not exceed the updated value of learning threshold, forward search operations continue. This way, the learning threshold acts as a guide to initiate the backtracking operations by comparing it with the accumulated heuristic learning.

In summary, backtracking is activated only when the accumulated heuristic learning exceeds the prescribed learning threshold. The learning threshold approach allows approximate solution with known quality to be determined, and more importantly the maximum amount of sacrifice is known before hand.

The algorithm of SLA*-TSP with learning threshold is described as follows:

Let S_i be the i^{th} state with its tour $P_i(1,2,...,i,1)$, where 1 is the city of origin and i is the last city of the tour. Its heuristic estimate h(i) is the minimum spanning tree of the remaining (n-i) cities. S_i is the goal state when i = n.

d(i,j) be the Euclidean distance between city i and city j.

H(i) be the estimated tour length for S_i, which consists of the tour p_i and h(i).

t = initial learning threshold

Step 0: Apply Delaunay triangulation algorithm to find neighbouring nodes for each city.

Step 1: Randomly select a city as the city of origin.

Step 2: Put the root state on the backtrack list called OPEN, and initialise learning threshold t' to the initial learning threshold (i.e. t' = t), and set search mode = forward.

Step 2: Call the top-most state of the OPEN list S_i. If S_i is the goal state, stop. Otherwise continue

Step 3: Find the $(i+1)^{th}$ city with min{[d(1,2)+d(2,3)+...+d(i-1,i)+d(i,i+1)+d(i+1,1)] +h(i+1)} from neighbouring cities of i; break ties randomly. If no neighbouring city of i can be found, remove S_i from the OPEN list if it is not the root state, and return to step 2.

Step 4: Let learning = min{[d(1,2)+d(2,3)+...+d(i-1,i)+d(i,i+1)+d(i+1,1)] +h(i+1)}-{[d(1,2)+d(2,3)+...+d(i-1,i)+d(i,1)] + h(i)}. If no learning has occurred (i.e. learning < 0), then add S_{i+1} to the OPEN list as the top-most state and return to step 2.

Step 5: If search mode = forward and heuristic learning has occurred but the value of the learning is less than or equal to the learning threshold t' (i.e. learning ≥ 0 and learning $\leq$ t'), then update t' by deducting the amount of learning that has taken place (i.e. t'= {t'- learning}) and replace h(i) with [d(i,i+1)+d(i+1,1) + h(i+1) - d(i,1)], add S_{i+1} to the OPEN list as the top-most state, and return to step 2.

Step 6: If search mode = forward and heuristic learning has occurred but the value of the learning is more than the learning threshold t' (i.e. learning ≥ 0 and learning > t'), then replace h(i) with [d(i,i+1)+d(i+1,1) + h(i+1) - d(i,1)] and set the search mode = backtrack. Remove S_i from the OPEN list if it is not the root state, and return to step 2.

Step 7: If search mode = backtrack and learning has occurred (i.e. learning > 0), then replace h(i) with [d(i,i+1)+d(i+1,1) + h(i+1) - d(i,1)], remove S_i from the OPEN list if it is not the root state, and return to step 2.

Step 8: If search mode = backtrack and no learning has occurred (i.e. learning ≤ 0), then set search mode = forward and initialise learning threshold to the initial learning threshold (i.e. t' = t), add S_i to the OPEN list as the top-most state, and return to step 2.

3.5.1 Examples

The same 8-city problem is used to demonstrate the working of SLA*-TSP with learning threshold. Tables 3.2, 3.3 and 3.4 show the search process of the 8-city problem with a learning threshold equals to 10%, 20% and 30%, respectively from the optimal solution. The optimal solution for the 8-city problem is 22300 (see Section 3.4.1). Therefore, the learning threshold prescribed for each of the three cases are t=2230, t=4460 and t=6690, respectively. To simplify the presentation, as in Table 3.1, only forward search operations are shown in the tables. It takes eleven forward search operations to find the optimal tour of (4,8,5,3,2,6,7,1,4) with length equal to 22300 when the learning threshold is at 10%. In Table 3.3, with the learning threshold is 20%, the solution found is also an optimal solution and it only takes five forward

operations. With learning threshold of 30%, although the solution found is not an optimal solution (tour length obtained is 24006), it is within the predicted range and it is 30% away from the optimal solution.

Iteration	Root state	Level-1	Level-2	Level-3	Level-4	Level-5	Level-6	Level-7
1	(4,4) * 13070	(4,1,4) 15891						
2	(4,4)* 15891	(4,1,4) 15891	(4,1,7,4) 15740	(4,1,7,8,4) 18548				
3	(4,4)* 18548	(4,1,4) 18548	(4,1,7,4) 18548	(4,1,7,8,4) 18548	(4,1,7,8,5,4) 18464	(4,1,7,8,5,3,4) 18409	(4,1,7,8,5,3,2,4) 16964	(4,1,7,8,5,3,2,6,4) 24122
4	(4,4)* 19348	(4,7,4) 19348	(4,7,8,4) 22569					
5	(4,4)* 20415	(4,5,4) 20415	(4,5,3,4) 20360	(4,5,3,2,4) 22171	(4,5,3,2,8,4) 20353	(4,5,3,2,8,6,4) 23384		
6	(4,4)* 20424	(4,8,4) 20424	(4,8,5,4) 21037	(4,8,5,3,4) 20982	(4,8,5,3,2,4) 21054	(4,8,5,3,2,6,4) 21678	(4,8,5,3,2,6,7,4) 21210	(4,8,5,3,2,6,7,1,4) 22300
7	(4,4)* 20740	(4,3,4) 20740	(4,3,5,4) 20360	(4,3,5,8,4) 20982	(4,3,5,8,2,4) 21650	(4,3,5,8,2,6,4) 22274	(4,3,5,8,2,6,7,4) 21806	(4,3,5,8,2,6,7,1,4) 28896
8	(4,4)* 21037	(4,5,4) 21037	(4,5,8,4) 21037	(4,5,8,7,4) 23310				
9	(4,4)* 21429	(4,1,4) 21429	(4,1,7,4) 21429	(4,1,7,8,4) 21429	(4,1,7,8,2,4) 21429	(4,1,7,8,2,5,4) 20744	(4,1,7,8,2,5,3,4) 16793	(4,1,7,8,2,5,3,6,4) 26402
10	(4,4)* 21445	(4,1,4) 21445	(4,1,7,4) 21445	(4,1,7,8,4) 21445	(4,1,7,8,5,4) 21445	(4,1,7,8,5,2,4) 21445	(4,1,7,8,5,2,3,4) 16643	(4,1,7,8,5,2,3,6,4) 26252
11	(4,4)* 22300	(4,8,4) 22300	(4,8,5,4) 22300	(4,8,5,3,4) 22300	(4,8,5,3,2,4) 22300	(4,8,5,3,2,6,4) 22300	(4,8,5,3,2,6,7,4) 22300	(4,8,5,3,2,6,7,1,4) 22300

Table 3.2: Search process with learning threshold equal to 10% of optimal solution (p=2230)

Iteration	Root state	Level-1	Level-2	Level-3	Level-4	Level-5	Level-6	Level-7
1	(4,4) * 13070	(4,1,4) 15891	(4,1,7,4) 15740	(4,1,7,8,4) 18548				
2	(4,4)* 18548	(4,1,4) 18548	(4,1,7,4) 18548	(4,1,7,8,4) 18548	(4,1,7,8,5,4) 18464	(4,1,7,8,5,3,4) 18409	(4,1,7,8,5,3,2,4) 16964	(4,1,7,8,5,3,2,6,4) 24122
3	(4,4)* 19348	(4,7,4) 19348	(4,7,8,4) 22569					
4	(4,4)* 20415	(4,5,4) 20415	(4,5,3,4) 20360	(4,5,3,2,4) 22171	(4,5,3,2,8,4) 20353	(4,5,3,2,8,6,4) 23384		
5	(4,4)* 20424	(4,8,4) 20424	(4,8,5,4) 21037	(4,8,5,3,4) 20982	(4,8,5,3,2,4) 21054	(4,8,5,3,2,6,4) 21678	(4,8,5,3,2,6,7,4) 22210	(4,8,5,3,2,6,7,1,4) 22300

Table 3.3: Search process with learning threshold equal to 20% of optimal solution (p=4460)

Iteration	Root state	Level-1	Level-2	Level-3	Level-4	Level-5	Level-6	Level-7
1	(4,4) * 13070	(4,1,4) 15891	(4,1,7,4) 15740	(4,1,7,8,4) 18548	(4,1,7,8,5,4) 18464	(4,1,7,8,5,3,4) 18409	(4,1,7,8,5,3,2,4) 16964	(4,1,7,8,5,3,2,6,4) 24122
2	(4,4)* 19348	(4,7,4) 19348	(4,7,8,4) 22569					
3	(4,4)* 20415	(4,5,4) 20415	(4,5,3,4) 20360	(4,5,3,2,4) 22171	(4,5,3,2,8,4) 20353	(4,5,3,2,8,6,4) 23384	(4,5,3,2,8,6,7,4) 22916	(4,5,3,2,8,6,7,1,4) 24006

Table 3.4: Search process with learning threshold equal to 30% of optimal solution (p=6690)

Table 3.3 is used to demonstrate how the learning threshold has influenced the forward search and backtracking processes. In this example, the learning threshold is set at 4460 and the city of origin is city 4. The selection of the next city to the tour is made from five neighbouring cities (1,3,5,7,8) of city 4. Therefore the corresponding states are: (4,1,4), (4,3,4), (4,5,4), (4,7,4) and (4,8,4). The selection of the state with minimum heuristic function is state (4,1,4) with the estimated value of 15891. A heuristic learning value of 2821 (the difference between the estimated tour length of the neighbouring state of (4,1,4) and (4,4)) has occurred. If no learning threshold has been set, then the algorithm will force backtracking to occur and the heuristic estimate of state (4,4) will be updated. However, with learning threshold, this allows the forward search process to continue. This is because the value of heuristic learning is less than the value of learning threshold. The estimated tour length of state (4,4) will be updated from 13070 to 15891. As heuristic learning has occurred, the learning threshold is now updated to 1639 (that is 15891-13070). From state (4,1,4), the neighbouring states to be generated is (4,1,7,4). Its estimated tour length is 15470 and this is not greater than the 15891, thus no heuristic learning takes place. Hence forward search process continues and the learning threshold remains at 1639.

From the state of (4,1,7,4), there are two possible neighbouring cities (cities 6 and 8) that can be added to the tour. Hence, the two corresponding front states are (4,1,7,6,4) and (4,1,7,8,4). The estimated tour length of state (4,1,7,6,4) is 23093 and that of state (4,1,7,8,4) is 18548. The state with the minimum value to be selected is state (4,1,7,8,4), with an estimated tour length of 18548. This value is greater than the estimated tour length of (4,1,7,4), thus heuristic learning is said to have occurred. The amount of heuristic learning in this case is 2808. This value is greater than the updated learning threshold value of 1639. Therefore backtracking is activated and the algorithm

backtracks to state (4,1,7,4), (4,1,4) and (4,4) and the estimated tour length is updated to 18548.

In row 2, the algorithm starts from (4,4) and selects (4,1,4), (4,1,7,4), (4,1,7,8,4). The learning threshold is reset to the initial learning threshold of 4460. No heuristic learning has taken place up to this state. From state (4,1,7,8,4), three possible front states are generated: (4,1,7,8,2,4), (4,1,7,8,5,4) and (4,1,7,8,6,4). It is found that state (4,1,7,8,5,4) with the estimated tour length of 18464 is the state with the minimum estimated tour length. As the estimated tour length of the front state is less than the estimated tour length of its parent state, no heuristic learning has occurred. The state (4,1,7,8,5,4) is now the front state. From this state, the search process continues. From the state (4,1,7,8,5,4), two possible front states are generated: (4,1,7,8,5,2,4) and (4,1,7,8,5,3,4). The state with the minimum estimated tour length is (4,1,7,8,5,3,4). Once again the estimated tour length of the front state (4,1,7,8,5,3,4) is 18409. This value is less than the estimated tour length of its parent state, therefore no heuristic learning has occurred and forward search process continues.

From the front state (4,1,7,8,5,3,4), two possible child states are generated: (4,1,7,8,5,3,2,4) and (4,1,7,8,5,3,6,4). At this point, the state with the minimum estimated tour length is state (4,1,7,8,5,3,2,4). This state with estimated tour length of 16964 is selected as the front state, and no heuristic learning has occurred up to this point. The forward search process continues and there is only one front state generated: (4,1,7,8,5,3,2,6,4). The estimated tour length of this state is 24122. This value is greater than the estimated tour length of its parent state (4,1,7,8,5,3,2,4). The heuristic learning that has taken place is equal to 7158, a value greater than the learning threshold of 4460. Therefore, the backtracking process is activated and the algorithm

backtracks to the root state and its estimated tour length is updated to 19348. With this new value, row 3 shows the next search process. The process continues until the solution of (4,8,5,3,2,6,7,1,4) with the estimated tour length equals to 22300 is found.

3.6 Conclusion

In this chapter, the main features of Search and Learning A* algorithm have been examined. The formulation of a traveling salesman problem into a state space problem is achieved by defining the state, state transition operator and state transition cost. The contribution of this chapter is the development of the state-space transformation process so that heuristic learning algorithm of SLA* can be applied to construct tours dynamically. This approach of constructing tour allows the tour configuration to change during the tour construction process. This is made possible by the backtracking and heuristic updating processes, which allow cities to be added and deleted during the tour construction processes. The application of heuristic evaluation function of the algorithm allows both local and global estimated distance information to be used. This way SLA*-TSP is not relying on local knowledge alone to build tour. This research has developed the SLA*-TSP approach which aims to overcome the greedy and myopic nature of traditional tour construction through the process of dynamic tour construction. The example presented in this chapter has shown that, the backtracking and forward search processes have repetitively led to the deletion and addition of cities from and to the tour through the consideration of both local and the global estimated distance information.

This chapter has also demonstrated that when the geometric properties of Delaunay triangulations is incorporated into the search strategy of SLA*-TSP, the performance

of SLA*-TSP is greatly enhanced through the reduction of solution space selection. This is because Delaunay triangulations identify only promising cities to be considered by a given state, and by ensuring that edges that will not lead to optimal solution are pruned during the search process.

The SLA*-TSP with learning threshold algorithm was also examined. This approach produces approximate solution of known quality, and it is particularly useful when the problem to be solved is too large with respect to computational resources.

CHAPTER 4: THE FACTORS INFLUENCING THE PERFORMANCE OF SLA*-TSP

4.1 Introduction

This chapter discusses the implementation of SLA*-TSP and the experimental results. SLA*-TSP was written in C++ and implemented using LEDA software library (Library of Efficient Data Types and Algorithms). LEDA is an object-oriented C++ class library consisting of various combinatorial and geometric data types and algorithms (Mehlhorn and Näher, 1999). It was applied in this research for defining data structures, computing Delaunay triangulation, minimal spanning tree and Euclidean distance between two nodes.

The factors influencing the performance of the SLA*-TSP approach will be investigated. The investigation will be based on computation experiments, which will be made on two sets of test problems: selected instances from the TSPLIB library (Reinelt, 1991) and randomly generated problems. In the computation experiments, only problems with x- and y-coordinates are tested. All distances are computed using the Euclidean distance function. The performance of SLA*-TSP algorithm with learning threshold will also be investigated with these problems.

This chapter is organised as follows. Section 2 discusses the issues in implementation of SLA*-TSP. In section 3, results of the test problems from TSPLIB library and randomly generated problems are discussed. The limitations and scope of the

experiments are also outlined in this section. In both categories of problems, SLA*-TSP with and without learning threshold will be applied. The presentations are based on the results obtained from each category of problems. Analysis and discussions of the factors influencing the performance of the algorithms are included in this section too. Finally, the conclusion follows in Section 4.

4.2 Implementation of SLA*-TSP using LEDA

The implementation of SLA*-TSP using LEDA involves the issues of data types. LEDA is a software library of combinatorial and geometric data types and algorithms (Mehlhorn and Näher, 1999). It provides a set of basic data types, such as string and stream, as well as parameterised data types, such as stack, array and list. It also provides geometric and graph algorithms (such as Delaunay triangulations and minimal spanning trees) that can be used in this research. It was selected based on its simplicity in data structure, its ability to be used with C++ compiler and a sizable collection of data types and computational geometric libraries that can be used. Since the implementation base of LEDA is C++, therefore SLA*-TSP is also implemented using C++.

The input of data set is based on x- and y-coordinates of the cities, and the data type *point* is used to denote the city in two-dimensional plane. It is represented by Cartesian coordinates (x,y). The LEDA data types used in the implementation of SLA*-TSP are parameterised. A parameterised graph denoted by *GRAPH* is of type *graph* (the data type *graph* consists of a list of nodes and a list of edges) whose nodes and edges contain additional user-defined data (Mehlhorn and Näher, 1999). A parameterised graph is used because it can dynamically allow information to associate with new

nodes and edges without restriction. This provides greater flexibility and more efficient access of information stored in the nodes and edges. In the implementation of SLA*-TSP, the declaration of parameterised graph is in the form of *GRAPH<point, int> G*. This creates an instance of G in which a string variable is associated with every vertex of G and an integer is associated with every edge of G. Delaunay triangulation is represented as *GRAPH<point, int> DT*. Minimal spanning tree is computed at the same time because it is a subgraph of Delaunay triangulation. This data type allows dynamic generation of Delaunay triangulation and computation of minimal spanning tree as the search process progresses. Another parameterised data type used in the program is list, *list<E>*, where E is the element in the list. It is used to store a sequence of items. In the implementation, list is used to store the sequence of cities in the partially completed tour.

The minimum heuristic estimate is stored using the parameterised data type stack, *stack<E>*, where E is the element in the stack. A stack uses the principle of last-in-first-out in which insertion and deletion of the elements only take place at the top of the stack. The elements stored in the stack is of data type *GRAPH<point, int>*. With forward search, the state with the minimum heuristic estimate is put at the top of the stack. This operation is performed using the operator S.push(). When in backtracking, the top element of the stack is deleted. The operation is performed using the operator, S.pop() which deletes and returns the new top element. This approach is used to keep track of backtracking and heuristic updating.

Heuristic estimate of a state is only saved after it has been updated during the backtracking process. The updated heuristic estimate is saved in a lookup table along with the partially completed tour. This allows the value of the heuristic estimate to be

referenced to the state in the search process. For states whose heuristic estimates are not found in the lookup table, they are calculated using the minimal spanning tree. The calculated value will be discarded if the proposed path was not used. However, if the proposed path is encountered again in the search process, the minimal spanning tree will be calculated again. Appendix A shows the listing of the program for the above implementation.

4.2.1 Pruning

One of the issues that need to be addressed in applying SLA*-TSP is to reduce the search space so that it is efficient. The following two approaches are used in the program to prune away states that are not promising in producing an optimal tour. In the first approach, when the child states are generated in the forward search, a dead-end state that will not lead to the generation of more subsequent child states is pruned from the search process. This can happen when the last city added in the partial tour does not have an unvisited neighbouring edge of Delaunay triangulation. When this occurs, the child state is pruned from the search space, as the forward search process cannot be continued.

The second approach is based on the rationale that the shortest closed path through n nodes is always a simple polygon. Therefore, an optimal TSP tour should not contain paths that intersect with one another, because intersection of paths will lead to a longer tour length. Thus in the second approach, when the generated child state results in a disjoint set of triangulation in the problem (a disjoint set of triangulation is one that divides the problem into two separate distinct sets of triangulations), then this child state will be pruned from the search space.

To illustrate the first approach, consider the example of 8-city problem in Figure 4.1. Assume that the current state is (4,7,4). According to Delaunay triangulation, three possible child states can be generated: (4,7,1,4), (4,7,6,4) and (4,7,8,4). The child state (4,7,1,4) will be pruned from the search space and only two states (4,7,6,4) and (4,7,8,4) will be generated as child states, because city 1 does not have any neighbouring city connected by the edges of Delaunay triangulation to allow forward search process to continue.

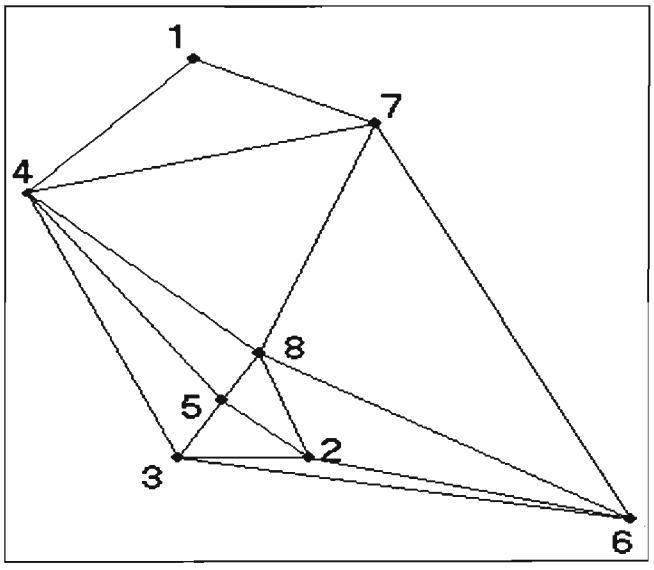

Figure 4.1: Delaunay triangulation of an 8-city problem

The following example explains how the second approach works using the same problem. Assume that the current state is (4,1,7,8,5,4). Then two possible child states are to be generated: (4,1,7,8,5,3,4) and (4,1,7,8,5,2,4). The child state (4,1,7,8,5,3,4) will be pruned, because it will generate the following two child states (4,1,7,8,5,3,2,4) and (4,1,7,8,5,3,6,4). Each of the child states consists of intersection paths that will not lead to an optimal tour.

4.3 Factors influencing the performance of SLA*-TSP

This section investigates the factors influencing the performance of the SLA*-TSP algorithm. The computation experiments were made on two sets of test problems: selected instances from the TSPLIB library (Reinelt, 1991) and randomly generated problems. All tests were run on a SUN SPARC server 20. Results and discussions from test problems obtained from TSPLIB will first be presented. This is followed by results and discussions on randomly generated problems.

4.3.1 Limitation and scope of the experiment

For memory saving purpose, only test problems from TSPLIB that have x- and y-coordinates are selected, where the distance between two cities is computed using the Euclidean distance function. The input in this form is required for the computation of Delaunay triangulations. This has restricted the choice of test problems that can be selected from TSPLIB. There are only ten problems in the set that have x- and y-coordinates with problem size less than one hundred nodes. These include problems with 14, 16, 22, 48, 51, 52, 70, 76 and 96 nodes (there are two problems with 76 nodes). Out of these ten problems, insufficient memory were encountered for problems that have more than twenty-two nodes. This problem is a result of using lookup table to store the states and its value of updated heuristic estimates at each backtracking step, and constrained the experiments to problems of a very small size.

Although the experiments were conducted on relatively small size problems, the aim is to analyse the results from these problems and investigate the search behaviour of the approach. Hence, the factors that influence the performance of SLA*-TSP algorithm may still be applicable to problems of larger sizes.

4.3.2 Test problems obtained from TSPLIB

This section presents the empirical test results from the selected problems in TSPLIB. In this section, the method is first presented. Then, results of SLA*-TSP without learning threshold are reported. Factors that influence the performance of SLA*-TSP will be examined as the results are reported. This is followed by presentation of results obtained from SLA*-TSP with learning threshold.

Method

The four problems that have been tested were named using the convention in TSPLIB – name follows by the number of cities in the problem. They are BURMA14, ULYSSES16, ULYSSES22 and OLIVER30. The empirical tests consist of running the algorithm for each starting point and the selection of the shortest tour as the optimal solution. For comparison purposes, solutions using nearest neighbour heuristic and Lin-Kernighan heuristic using the Concorde program (Applegate et al, 1998) were calculated.

SLA*-TSP was first applied to these problems without the learning threshold, then it was applied with the learning threshold. The learning threshold was set at 10%, 30%, 50%, 70% and 90% of the optimal solution.

Results and discussions

SLA*-TSP without learning threshold

This section presents the empirical test results in terms of computation time and the solution obtained. Table 4.1 shows the results of the four problems using SLA*-TSP, nearest neighbour heuristic and the Lin-Kernighan heuristics. The tour length and the

CPU time (in seconds) that were obtained using each method are displayed in the table too. The results in tour length show that the heuristic learning algorithm of SLA*-TSP produces better solution than the nearest neighbour heuristic, and is better or as good as that obtained from the Lin-Kernighan heuristic. However, SLA*-TSP performs poorly in computation time compared with both nearest neighbour and Lin-Kernighan heuristics.

Problem	SLA*-TSP		Nearest-neighbour		Lin-Kernighan	
	Tour length	CPU time (in sec)	Tour length	CPU time (in sec)	Tour length	CPU time (in sec)
BURMA14	30.8785	43	37.7108	0	31.4536	0.05
ULYSSES16	74.1989	782	88.5888	0	74.1989	0.20
ULYSSES22	75.401	5222	90.58963	0	76.6657	0.40
OLIVER30	423.741	2033	586.1363	0	423.741	0.17

Table 4.1: Results of four test problems

The poor performance of SLA*-TSP in computation time can be attributed to dynamic tour construction and inefficient data retrieval in the program. In SLA*-TSP approach, a tour is constructed by allowing addition and deletion of cities to and from the tour. This dynamic tour construction continuously updates along the search process. On the other hand, nearest neighbour heuristic, which is a greedy approach, does not change the tour configuration. This is because in the greedy approach, the part of the tour already built remains unchanged during the tour construction process. Therefore, it can be seen that dynamic construction of the tour can lead to a better solution. However, constant updating of the partial tour through the process of backtracking and updating of heuristic estimates can result in a much higher computation cost. The use of the lookup table in keeping track of the updated heuristic estimates and its state may have contributed toward the longer computation time. The size of the lookup table increases as the search process progresses as more heuristic estimates are updated during the

backtracking processes. Although binary search is used in the program to retrieve the value of the updated heuristic estimates from the lookup table, the retrieval time can still grow substantially.

The performance of SLA*-TSP is not entirely influenced by the number of nodes alone. The heuristic estimate is another important factor to consider. This factor can be measured as the ratio of the initial heuristic estimate of the root state to the optimal solution. If the ratio is high, then the quality of the heuristic estimate is good, and it can be expected that fewer backtracking and heuristic updates are necessary. Otherwise, from a lower base, more backtracking and heuristic updates will be required before the optimal solution is found. Therefore if the initial heuristic estimate is too much under-estimated, a longer computation time is to be expected. Table 4.2 shows a comparison of the number of heuristic updates in relation to the quality of the heuristic estimate, which is measured as the ratio of initial heuristic estimates of the root state to the tour length of the optimal solution. Column 2 shows the number of heuristic updates required and column 3 shows the ratio: 0.65, 0.63, 0.61 and 0.80 respectively.

Problem	Number of heuristic updates	Ratio = initial heuristic estimate of the root state/optimal solution
BURMA14	5601	0.65
ULYSSES16	101005	0.63
ULYSSES22	425565	0.61
OLIVER30	96446	0.80

Table 4.2: Ratio of initial heuristic estimate of root state to the tour length

In Table 4.2, the number of heuristic updates required by OLIVER30 is 96446, and the number of heuristic updates required by ULYSSES16 and ULYSSES22 (which have fewer nodes compare to OLIVER30) are 101005 and 425565 respectively. The quality

of the heuristic estimates of the latter two problems (0.63 and 0.61, respectively) is relatively poor compared to OLIVER30, which has a ratio of 0.80. From the CPU time in Table 4.1, one can see that it takes a shorter time for OLIVER30 to reach optimal solution.

When the initial value of heuristic estimate is high, then it will take fewer heuristic updates to reach the optimal solution. Therefore, one can conjecture that the quality of the heuristic estimate can influence the performance of the algorithm.

SLA*-TSP with learning threshold

This section reports the results of the above four problems when SLA*-TSP with learning threshold is applied. Table 4.3 shows the results with 10%, 30%, 50%, 70% and 90% of learning thresholds. Column 2 shows the levels of the learning threshold expressed in percentage form as well as the actual value of learning threshold applied in each level. Column 3 shows the CPU time (in seconds) required to find the solution. Column 4 shows the number of heuristic updates it takes to reach the solution, and column 5 shows the percentage of saving in heuristic updates, which is calculated as $(1-\frac{NumberOfHeuristicUpdateWithLearningThreshold}{NumberOfHeuristicUpdateWithoutLeaningThreshold})$. The solution found for each learning threshold is not displayed. Instead, the penalty on the solution, which is expressed as $(1+\frac{SolutionFound-OptimalSolution}{OptimalSolution})$, is shown and is given in column 6. In Table 4.3, the first row of each problem (rows with learning threshold = 0%) has been included for comparison purposes.

Problems	Leaning threshold		CPU time (in sec)	Number of heuristic updates	Saving in heuristic updates with learning threshold	Penalty on the solution
	%	actual value				
BURMA14	0%	0	43	5601	-	1.00
	10%	3.09	9	1621	71%	1.03
	30%	9.26	4	34	99%	1.28
	50%	15.44	0	51	99%	1.28
	70%	21.61	0	0	100%	1.15
	90%	27.79	0	0	100%	1.15
ULYSSES16	0%	0	782	101005	-	1.00
	10%	7.42	286	36282	64%	1.01
	30%	22.26	19	1739	98%	1.19
	50%	37.10	2	168	100%	1.07
	70%	51.94	2	168	100%	1.07
	90%	66.78	2	168	100%	1.07
ULYSSES22	0%	0	5222	425565	-	1.00
	10%	7.54	584	40191	91%	1.02
	30%	22.62	4	122	100%	1.04
	50%	37.70	0	9	100%	1.09
	70%	52.78	0	1	100%	1.07
	90%	67.86	0	0	100%	1.07
OLIVER30	0%	0	2033	96446	-	1.00
	10%	42.37	590	19096	80%	1.09
	30%	127.12	9	232	100%	1.21
	50%	211.87	11	443	100%	1.16
	70%	296.62	11	507	99%	1.13
	90%	381.37	11	507	99%	1.13

Table 4.3: Results of all four problems when learning threshold is applied

It is apparent that when the learning threshold is applied, the number of heuristic updates required to reach the solution is significantly reduced compared with the case where learning threshold is not applied. The savings obtained by the four problems range from 64% to 91% and 98% to 100% when learning threshold of 10% and 30% are respectively applied. With a 30% learning threshold, the saving range from 90% to 100%. It is interesting to compare the saving of heuristic updates and penalty on the solution. At 10% learning threshold, for BURMA14, the saving is 71%, but the penalty on the solution is only 3%. Similarly for problem ULYSSES16, the saving is 64%, and the penalty on the solution is only 1%. Similar trends can be observed for problems ULYSSES22 and OLIVER30. It can be seen that the number of heuristic updates

decreases as the learning threshold becomes larger. In each case, the saving in heuristic updates is quite significant while the penalty on solution is relatively small. SLA*-TSP with learning threshold could produce an approximate solution that is within the range of the specified threshold from optimal solution. Results from the experiment show that the penalty on the solution is within the range of the learning threshold set. In each case the number of heuristic updates is reduced with the increase in learning threshold. The saving of the heuristic updates can be attributed to the reduced frequency in responding to the backtracking process, which is not invoked until the cumulative heuristic update reaches the prescribed learning threshold. Therefore, one can conjecture that if an approximate solution is sufficient for a problem, then SLA*-TSP with learning threshold may be applied with reasonable computation time and computer memory resources. This approach may be useful when one is faced with large and complex problems that require excessive computation resources.

Another observation that can be made from solutions with learning threshold is that the solution seems to plateaus after a certain range of learning threshold. Figure 4.2 shows the graph on penalty on the solution with respect to learning thresholds. It can be seen that the quality of the solution deteriorates as the learning threshold increases. However, the solution does improve slightly and then plateau off as the learning threshold increases.

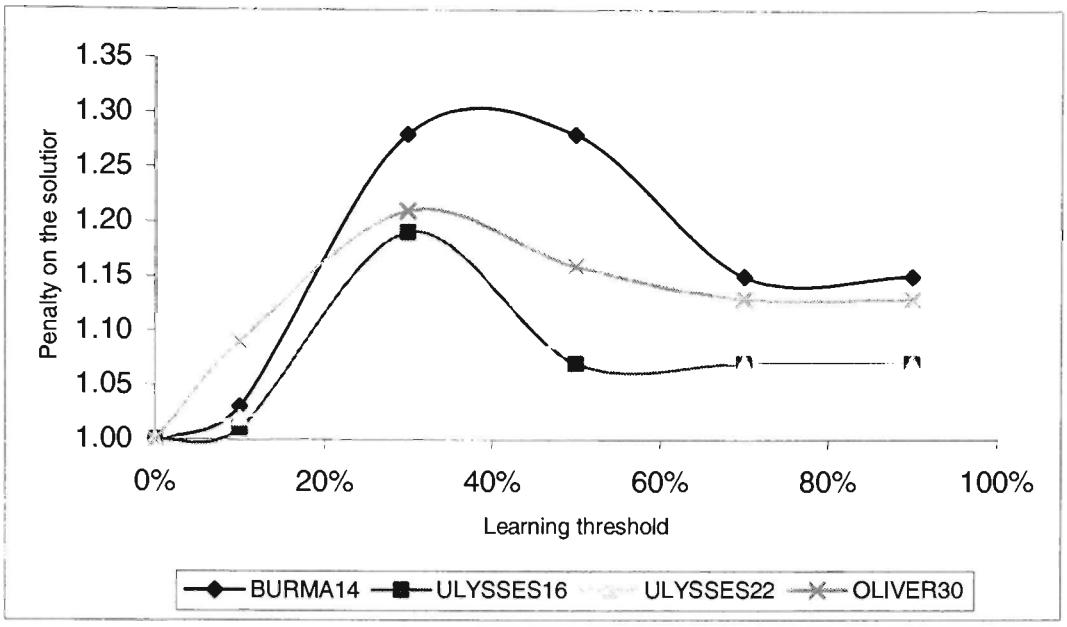

Figure 4.2: Graph showing penalty on the solution in term of learning threshold

In the current implementation of the program, only small sized problems can be solved within reasonable time. Large sized problems could not be solved within reasonable computation time due to inefficient memory handling. Table 4.4 shows the results of running SLA*-TSP with learning threshold for seven test problems from TSPLIB. The optimal solution for each problem is given by TSPLIB. It can be seen that the computation time has been greatly reduced and the solution found is within the range of the learning threshold. The results from Table 4.4 indicate that the solution plateaus when the learning threshold reaches around 70% of the optimal solution. By incorporating learning threshold to the SLA*-TSP approach, the algorithm was able to find an approximate solution to the problem with a desired range of certainty.

In summary, the SLA*-TSP with learning threshold approach can improve the computation time for large-sized problems at the sacrifice of the optimal solution. The advantage in using this approach is the maximum amount of sacrifice is known before hand.

Problem	Learning threshold	CPU time (in sec)	Number of heuristic updates	Penalty on the solution
ATT48	30%	17	157	1.27
	50%	12	130	1.30
	70%	15	135	1.29
	90%	15	135	1.29
EIL51	20%	34	356	1.15
	30%	41	691	1.12
	50%	2	0	1.15
	70%	2	0	1.15
	90%	2	0	1.15
BERLIN52	30%	224	3699	1.22
	50%	55	832	1.33
	70%	10	112	1.31
	90%	6	49	1.33
EIL76	30%	169	4055	1.13
	50%	95	2327	1.13
	70%	96	2346	1.13
	90%	96	2346	1.13
RAT99	30%	10	10	1.27
	50%	11	6	1.28
	70%	10	0	1.27
	90%	10	0	1.27
EIL101	30%	13	22	1.22
	50%	14	45	1.22
	70%	11	0	1.24
	90%	11	0	1.24
LIN105	55%	189	7393	1.37
	60%	192	7512	1.40
	70%	191	7516	1.40
	90%	190	7388	1.37

Table 4.4: Computation results for selected TSPLIB problem instances

4.3.3 Randomly generated problems

The aim of this experiment is to investigate the performance of SLA*-TSP in relation to the pattern in which the nodes are distributed in the Euclidean plane. The investigation will be conducted using randomly generated problems. The clusters can be dispersed and well separated, in which case the distance between different clusters (called inter-cluster distance) can be significant. On the other hand, the clusters can be close to one another, in which case the inter-cluster distance is insignificant.

In this section, the method is first presented. Then, the results of SLA*-TSP without learning threshold are reported. Discussion of results will also be included. This is followed by presentation of results using SLA*-TSP with learning threshold.

Method

The test approach is adapted from Laporte et al (1996), and the test problems consist of nodes located within a (0,100) square. The square is divided into 16 equal rectangles (see Figure 4.3). Six nodes are randomly generated within each rectangle according to a uniform distribution. The nodes were selected from four rectangles within the square and each test problem consists of twenty-four nodes. Each problem is named using the number matching each rectangle in the square. For example, p1_4_16_13 refers to nodes obtained from rectangles 1, 4, 16 and 13.

4	8	12	16
3	7	11	15
2	6	10	14
1	5	9	13

Figure 4.3: Structure of problems tested

Five problems have been selected in this experiment: p1_6_11_16, p3_7_11_15, p1_2_3_4, p1_8_9_16, p1_4_16_13. Each problem shows a different characteristic in which the clusters are located in the square. For example, the nodes in problems p1_4_16_13 and p1_8_9_16 show a characteristic of clusters that are dispersed and well separated. On the other hand, the clusters in problems p1_2_3_4, p3_7_11_15 and p1_6_11_16 are located close to one another. In terms of distance between clusters, problems p1_4_16_13 and p1_8_9_16 both demonstrate significant inter-cluster distance, comparing to the other three problems.

Results and discussions

SLA*-TSP without learning threshold

This section presents the test results in terms of CPU time, number of heuristic updates and the quality of the heuristic estimate, which is expressed as the ratio of the initial heuristic estimate of the root state to the optimal solution. The results shown were averaged over ten different instances for each type of problem. Table 4.5 shows the results obtained when SLA*-TSP without learning threshold is applied. Column 2 shows the CPU time in second, column 3 shows the number of heuristic updates required, and column 4 shows the ratio.

Problem	CPU time (in sec)	Number of heuristic updates	Ratio = initial heuristic estimate of the root state/optimal solution
p1_6_11_16	4675	321201	0.5896
p3_7_11_15	4747	295200	0.6547
p1_2_3_4	3909	257204	0.6280
p1_8_9_16	474	26527	0.6908
p1_4_16_13	141	7159	0.7564

Table 4.5: Results without learning threshold

It is obvious that the ways nodes are grouped in different clusters can influence Delaunay triangulations. Minimal spanning tree is a subset of Delaunay triangulation (Aurenhammer, 1991), thus the behaviour of clustering can influence the quality of the heuristic estimate, which is computed using minimal spanning tree. Figure 4.4 and 4.5 show Delaunay triangulations for problems p1_4_16_13 and p1_2_3_4 respectively. These two problems were selected to contrast the Delaunay triangulations formed. The nodes in problem p1_4_16_13 are grouped into distinct and well-separated clusters. The Delaunay triangulations produced can be described as 'wide' and 'fat'. On the other hand, the clusters in problems p1_2_3_4 are close to one another, and the

Delaunay triangulations produced can be described as 'elongated', 'narrow' and 'skinny'. The descriptions used to described Delaunay triangulations as 'fat' and 'skinny' follow the convention used in Aurenhammer (1991), de Berg et al (1997) and O'Rourke (1998).

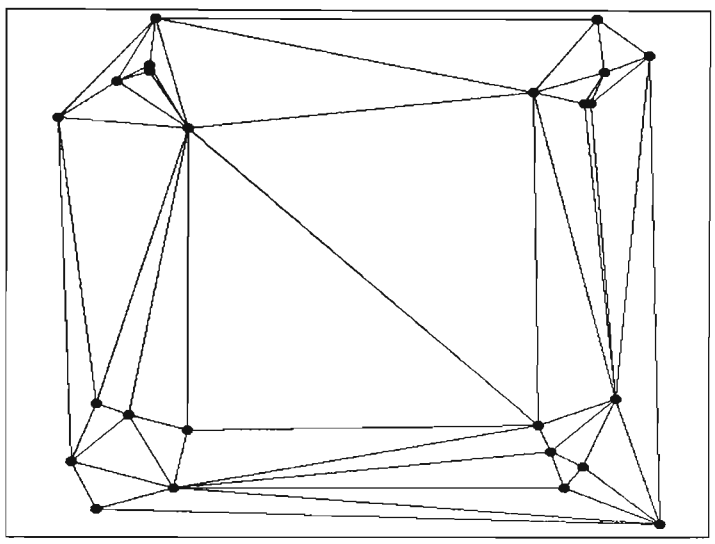

Figure 4.4: Delaunay triangulation for p1_4_16_13

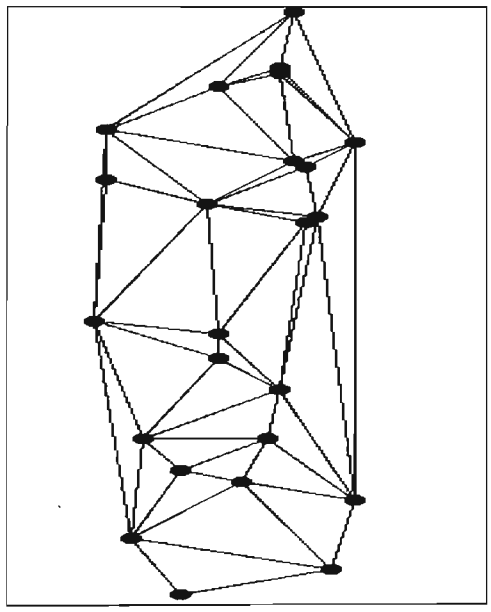

Figure 4.5: Delaunay triangulation for p1_2_3_4

The last column of Table 4.5 shows that the ratio for problems p1_4_16_13 and p1_8_9_16 is higher, comparing to the other three problems. In addition, the number of heuristic updates for these two problems is comparatively smaller comparing to the

other three problems. This result can be explained by the way the nodes are grouped into clusters. The clusters in problems p1_4_16_13 and p1_8_9_16 are more dispersed and well-separated. In contrast, the clusters in each of the problems p1_2_3_4, p1_6_11_16 and p3_7_11_15 are situated near to one another. In problems where clusters are distinct and well-separated, the edges that connect different clusters could be longer. These long edges between the clusters would form part of the minimal spanning tree, because minimal spanning tree must connect all nodes. Therefore, when the clusters are dispersed and located far from one another, it will result in a higher heuristic estimate. On the other hand, in problems where clusters are situated near one another, the edges between clusters are shorter. This could result in a lower value of heuristic estimate. When the heuristic estimate is comparatively low, it will take more heuristic updates and backtracking to reach the optimal solution. Therefore, one can conjecture that SLA*-TSP approach is suitable for problems that exhibit distinct and well-separated clusters. This is because long edges between the clusters will always form part of the edges in minimal spanning tree, and it will result in a higher value of initial heuristic estimate of the root state, which can lead to better performance.

Reinelt (1992) shows that Delaunay triangulation can provide a better candidate set for problems in which nodes are located in several clusters. This is because edges that connect different clusters are included in the triangulations. This research uses Delaunay triangulations to define the candidate edge set and the results from this experiment are consistent with the findings from Reinelt (1992).

SLA*-TSP with learning threshold

This section presents the results of the above five problems when SLA*-TSP with learning threshold is applied. The learning threshold is applied at five different levels:

10%, 30%, 50%, 70% and 90%. The results presented here were averaged over ten different instances for each problem, and are reported in terms of number of heuristic updates, computation time and the quality of the heuristic estimate. Table 4.6 shows the number of heuristic updates and saving in heuristic updates when learning threshold is applied. For each problem, the first column shows the number of heuristic updates, and the second column shows the saving achieved (expressed in percentage). Tables 4.7 and 4.8 show the CPU time and the solution quality respectively. In Tables 4.6 and 4.7, results without learning threshold (i.e. rows with learning threshold = 0%) are included for comparison purposes.

The quality of the solution did not suffer (penalty on the solution is zero) when 10% of the learning threshold is applied, whereas the savings in heuristic updates range from 60% to 77%. When 30% of the learning threshold is applied, penalty on the solution range from 2% to 5% with the savings ranging from 89% to 98%. In each problem, 100% of saving is achieved in terms of heuristic updates when the learning threshold reaches 50%. In terms of penalty on the solution, it can be seen that the penalty of solution ranges from 8% to 17% at 50% of learning threshold. Therefore, in each case, the saving in terms of heuristic updates and computation time are quite significant.

Learning Threshold	Heuristic updates									
	p1_2_3_4		p1_6_11_16		p3_7_11_15		p1_4_16_13		p1_8_9_16	
	Number	Saving	Number	Saving	Number	Saving	Number	Saving	Number	Saving
0%	257204	-	321201	-	295200	-	7159	-	26527	-
10%	59417	77%	74912	71%	70190	76%	2339	67%	10695	60%
30%	5812	98%	4550	98%	2971	99%	164	98%	2855	89%
50%	83	100%	300	100%	276	100%	243	97%	1017	96%
70%	20	100%	49	100%	59	100%	10	100%	114	100%
90%	6	100%	42	100%	58	100%	2	100%	41	100%

Table 4.6: Number of heuristic updates and saving (expressed in %) with different levels of learning thresholds

Learning Threshold	CPU time (in sec)				
	p1_2_3_4	p1_6_11_16	p3_7_11_15	p1_4_16_13	p1_8_9_16
0%	3909	4675	4747	141	474
10%	841.9	1132.4	1116	47.4	197.7
30%	116.7	87.3	67.3	5.4	61.5
50%	2.5	7.6	6.6	6.9	24.2
70%	0.6	1.7	1.5	0.6	2.8
90%	0.4	1.5	1.5	0.5	1.2

Table 4.7: CPU time with different levels of learning thresholds

Learning Threshold	Solution Quality				
	p1_2_3_4	p1_6_11_16	p3_7_11_15	p1_4_16_13	p1_8_9_16
10%	1.00	1.00	1.00	1.00	1.00
30%	1.04	1.02	1.02	1.03	1.05
50%	1.11	1.08	1.17	1.09	1.10
70%	1.08	1.11	1.18	1.07	1.09
90%	1.11	1.08	1.19	1.10	1.09

Table 4.8: Quality of solution with different levels of learning thresholds

Figures 4.6, 4.7 and 4.8 show the relationship between different levels of learning thresholds with respect to the number of heuristic updates, CPU time and penalty on the solution respectively. The graph shows that computation time and heuristic updates decrease significantly with the increase in learning threshold. The quality of the solution initially deteriorates when the learning threshold is applied. However, the solution generally plateaus when the learning threshold reaches about 50% of the optimal solution (with the exception of p1_4_16_13). The results again show that SLA*-TSP with learning threshold approach can improve the computation time if the exact solution is not required.

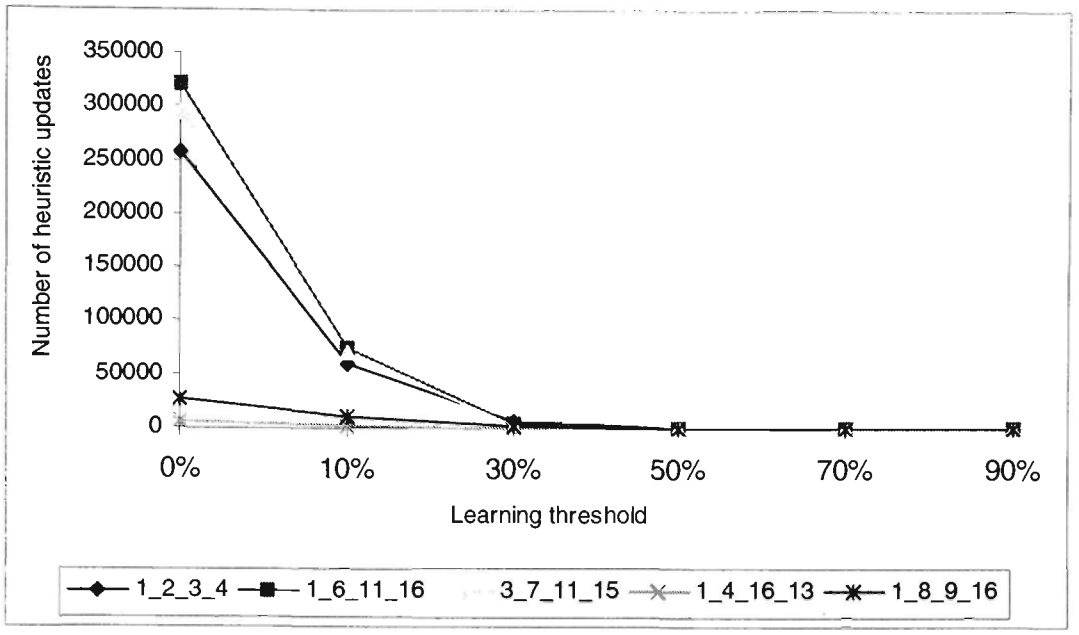

Figure 4.6: The number of heuristic updates vs. learning threshold

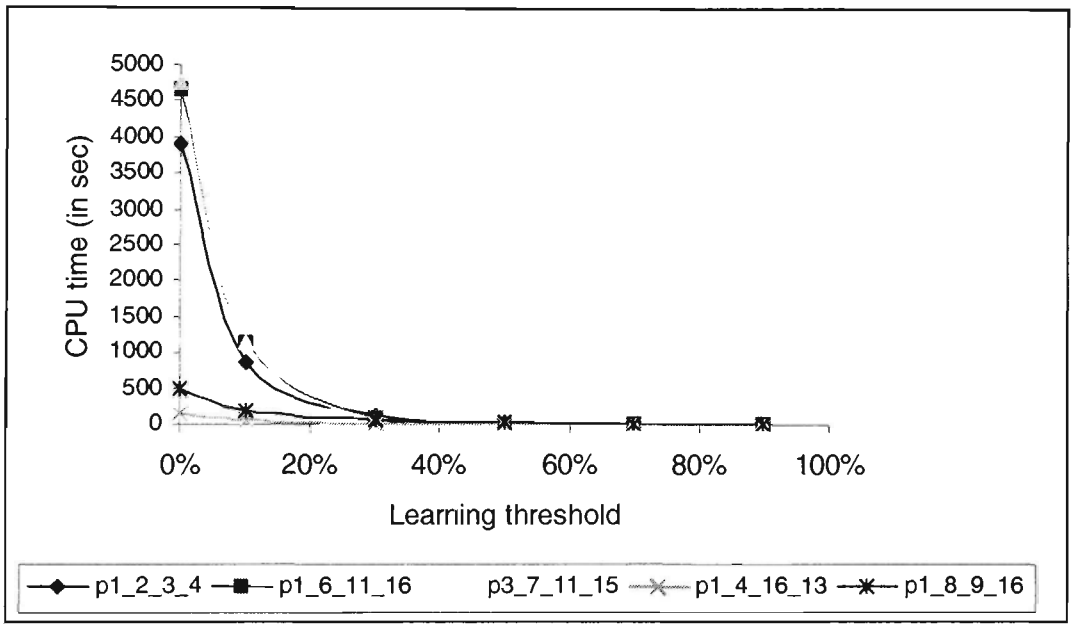

Figure 4.7: Performance of CPU time vs. learning threshold

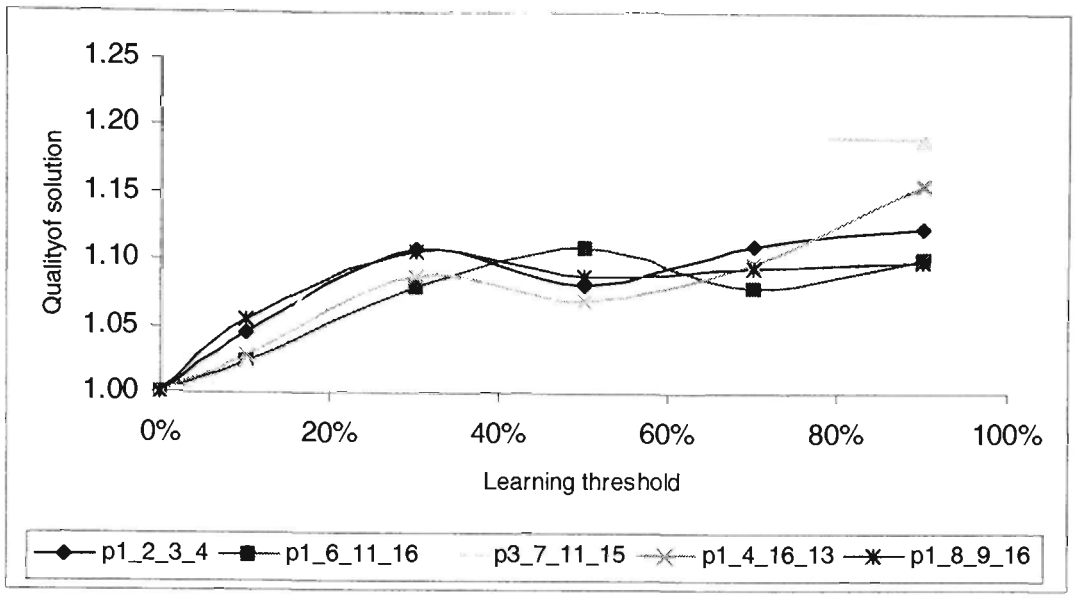

Figure 4.8: Quality of solution vs. learning threshold

4.4 Conclusion

In this chapter, the process of implementation of the SLA*-TSP approach is discussed. The SLA*-TSP algorithm has been implemented using LEDA in C++. The major weakness of the current implementation is inefficient memory handling. It is believed that if the data structures and the current design of the lookup table can be improved, then problems of larger size can be solved. Further investigation needs to carry out to improve the design of data structures so that memory can be handled more efficiently. One possible improvement to the memory management is to use the tree structure, and not the lookup table, to store the state and the value of the updated heuristic estimate.

The results show the main factor that influences the performance of SLA*-TSP is the value of the initial heuristic estimate. If the initial value of the heuristic estimate of the root state is close to the optimal solution, then fewer backtracking and heuristic updates will be required to reach the optimal solution. The way nodes are grouped into clusters in the Euclidean plane also influences the quality of the heuristic estimate. A

better performance of SLA*-TSP can be achieved for problems that exhibit distinct and well-separated clusters, because this type of problem produces a higher quality of heuristic estimate. Therefore, the closer the value of the heuristic estimate is to the optimal solution, the better the performance of SLA*-TSP.

Finally, the results also show that SLA*-TSP with learning threshold approach can improve the computation time for large-sized problems at the sacrifice of the quality of the solution. By incorporating learning threshold to the SLA*-TSP approach, the algorithm is able to find an approximate solution to the problem with a known quality. The advantage of this approach is that the maximum amount of sacrifice is known before hand. This feature is particularly important in practical situations when the exact solution is not required but the speed of computation is critical.

CHAPTER 5: A RESTRICTIVE SEARCH APPROACH

5.1 Introduction

The SLA*-TSP approach developed in Chapter 3 allows the tour configuration to change during the tour construction process. The use of Delaunay triangulation as a search strategy allows promising edges to be identified from which the state transition operator takes edges with priority. However, this approach still leads to a comparatively large search space. This is evidenced in the small sized problems that can be solved in Chapter 4. This chapter discusses the implementation of a restrictive search approach that could further reduce the number of candidate edges during the search process. This represents an improvement over the search strategy using the Delaunay triangulations, which was defined in Chapter 3. The approach is to define the candidate edge set using edges from only one triangle selected from Delaunay triangulations. The criteria for selecting the triangle are based on the concept of proximity in Voronoi diagram, the direction of the tour and the search direction of the triangle.

This chapter is organised as follows. Section 2 explains the rationale behind the factors that can be used to further reduce the search space. Section 3 presents the restrictive SLA*-TSP algorithm that incorporates the restrictive search strategy. Three examples are provided to demonstrate the implementation of this approach in Section 4, and Section 5 concludes the chapter.

5.2 Identifying proximity and utilising knowledge of direction in tour construction

This section examines the rationale behind the development of a restrictive search approach which is used as a constrained search strategy to further reduce the search space in SLA*-TSP approach. The method selects only one triangle from Delaunay triangulations using the proximity properties of Voronoi diagram, the travelling direction of the tour and the search direction of the triangle. Before these factors are examined, the following terms will be used throughout the chapter. An *external* node is a node that is on the convex hull, and an *internal* node is one that is not on the convex hull. A node is located to the *right* side of the current node if its x-coordinate is larger than the x-coordinate of the current node, and a node is to the *left* side of the current node if its x-coordinate is smaller than the x-coordinate of the current node.

The proximity property of Voronoi diagram presents another useful concept in reducing the search space. As pointed out in Chapter 3, Delaunay triangulation and Voronoi diagram are dual structures. This means both contain the same information although they are represented in different forms (O'Rourke, 1998). As discussed previously, Delaunay triangulation is used to identify promising candidate edges during the search process, therefore Voronoi diagram should contain similar information that can be used to identify the promising candidate edge set. A Voronoi diagram is a computational geometric structure that represents proximity information about a set of points (Aurenhammer, 1991). The Voronoi diagram divides the nodes into a set of polygons (called sites) of which the boundaries are perpendicular bisectors between two nodes. The Voronoi polygon around each site consists of nodes that lie closer to that site than to any other site. Finding the nearest neighbouring node means

identifying the boundary of the Voronoi region, because any point that lies inside the region is its nearest neighbour (O'Rourke, 1998). Thus the boundary of the Voronoi region can be used to identify nodes that are closer to that site than to others. Using this principle, we can construct Voronoi diagram with only external nodes. This way, the proximity information can be used to identify internal nodes that are nearer to one particular external node than another.

A reduced candidate edge set called *proximity candidate edge set* can be defined as nodes that are located both in the candidate edge set derived from Delaunay triangulation (as explained in Chapter 3) and candidate edge set derived from Voronoi diagram. The intersection of these two sets is the *proximity candidate edge set*, which can be used to select candidate neighbouring city to be included in the tour with priority. If there is no other internal node in the same Voronoi region as the external node, then the intersection of the two sets is null. In this case, let the proximity candidate edge set to be the same as the candidate edge set from Delaunay triangulation for that node. This procedure is only applied to external nodes because the Voronoi diagram is constructed using the external nodes only. For all other internal nodes, the proximity candidate edge set is the same as that defined in Chapter 3, which is based on the edges derived from Delaunay triangulation. The procedure of finding the proximity candidate edge set is given in Figure 5.1.

procedure find_proximity_candidate_edge_set

begin

1. Apply the algorithm of Delaunay triangulation to all nodes in the problem. For each node x, if node y is connected to node x via the edge of Delaunay triangulation, then y is included in the candidate set for node x. Call this candidate edge set D_x.
2. Apply the algorithm of the Voronoi diagram to find the Voronoi regions for external nodes only.
3. For each external node x on convex hull, if internal node y is in the same Voronoi region as x, then y is included into the Voronoi candidate edge set for x. Call this candidate edge set V_x.
4. If no other internal node is in the same Voronoi region as x, then let $V_x = \varnothing$.
5. For each external node x, find the intersection of D_x and V_x, $\{D_x \cap V_x\}$. Call this the *proximity candidate edge set* P_x.
6. If $P_x = \varnothing$, then let $P_x = D_x$.
7. For all other internal node y, let $P_y = D_y$.

end

Figure 5.1: Find_proximity_candidate_edge_set procedure

Figure 5.2 shows an 8-city problem, which will be used to explain how the procedure of find_proximity_candidate_edge_set works. Figure 5.3 shows the Voronoi diagram that is constructed using external nodes (1, 4, 3, 6, 7). Figure 5.4 shows the diagram that combines Figures 5.2 and 5.3. In this diagram, the boundary of Voronoi region is displayed in bold. To demonstrate how the above procedure works, consider node 3. Based on the edges of Delaunay triangulation its candidate edge set is $D_3 = \{2, 4, 5, 6\}$. Its candidate edge set using the proximity approach of Voronoi diagram as explained above is $V_3 = \{5, 2, 8\}$. Therefore the proximity candidate edge set $P_3 = \{D_3 \cap V_3\} = \{2, 5\}$. Nodes 4 and 6 are in the candidate edge set derived from Delaunay triangulation, however they are not in the same Voronoi region as node 3. Although node 8 is in the same Voronoi region as node 3, but it is not in the candidate edge set D_3, therefore it will be discarded. This shows that only nodes 2 and 5 will form the proximity candidate edge set for node 3, and the number of promising neighbouring nodes to be selected during the state transition process is reduced from four in D_3 to

two in P_3.

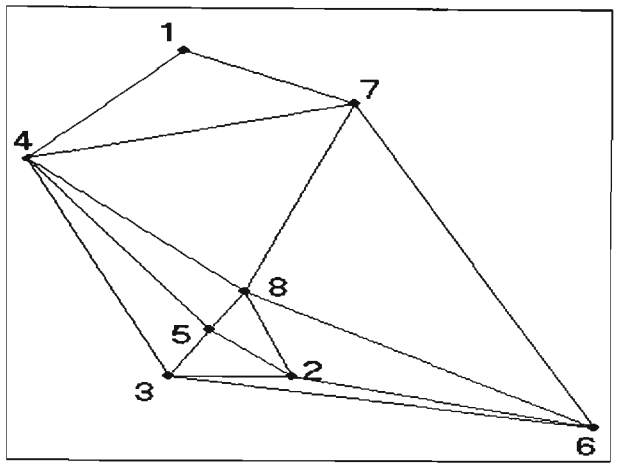

Figure 5.2: Delaunay triagulation of an 8-city problem

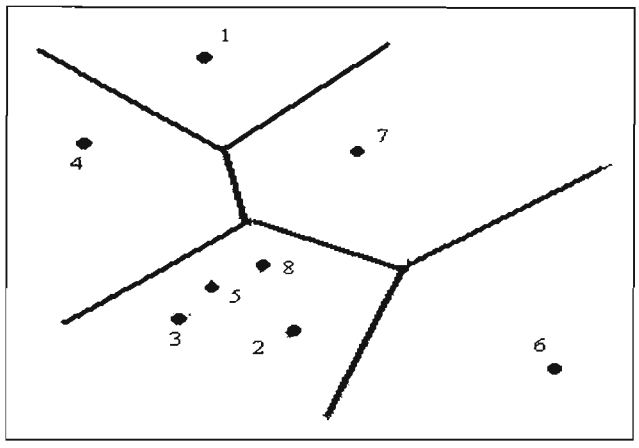

Figure 5.3: Voronoi diagram for external nodes only

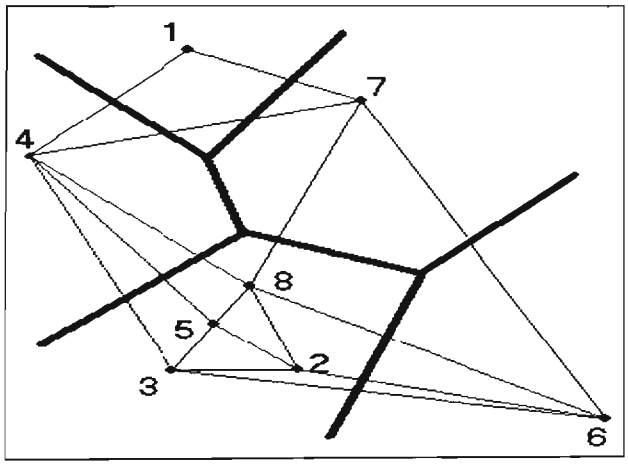

Figure 5.4: Combined Delaunay triagulation and Voronoi diagram

The second factor to consider is the travelling direction of a given tour. It is known that in any optimal TSP tour, nodes that are located on the convex hull are visited in

the order in which they appear on the convex hull boundary. Otherwise, it will contain intersection of paths that will not lead to optimal solution (Lawler et al, 1985; Reinelt, 1994; Cesari, 1996). One of the properties of Delaunay triangulation is, its boundary forms the convex hull (Aurenhammer, 1991). Thus for a node on the boundary of convex hull, it is joined to two other neighbouring nodes on the convex hull through the edges of Delaunay triangulation. Using the search strategy as defined in the SLA*-TSP approach in Chapter 3, both edges are included for consideration during the search process. However, it is only necessary to consider only one of these two nodes, depending on the direction the tour travels. If the tour travels in a clockwise direction, then it is more likely that this node will go to the node on its left side, and not to the node on its right side. Similarly if the tour travels in an anti-clockwise direction, then it will travel to the node on its right side, and not to its left. In this case, the node to its left can be pruned from the search space.

The example in Figure 5.2 is used to illustrate the above idea. In this example, the convex hull is made up of nodes (1, 4, 3, 6, 7). Each node on the convex hull is connected to two other nodes on the boundary of the convex hull. At node 3, it is joined to nodes 4 and 6 through the edges of Delaunay triangulation. Node 4 has its x-coordinate smaller than node 3, and is to the left of node 3; node 6 is hence located to the right of node 3. As explained above, one of these nodes can be pruned from the search space depending on the direction the tour travels. If the direction the tour travels is anti-clockwise, then node 3 is likely to travel to node 6, and it is not necessary to consider node 4. On the other hand, if the direction of the tour is clockwise, node 3 can only travel to node 4, rather than node 6. This way only one of the nodes needs to be selected during the search process and the other node can be excluded from the search space.

As discussed above, some nodes can be discarded from the search space depending on the direction the tour travels. The restrictive search approach proposed in this chapter is designed to select the candidate neighbouring city via the edges of only one triangle from the Delaunay triangulations. The triangle to be selected depends on the direction the tour travels. The following rule can generally be applied to assist in identifying the triangle. Before the procedure is examined, the following terms will be defined first. A node can be divided into *quadrants*, and they are labelled as first, second, third and fourth in a clockwise direction (see Figure 5.5). A *right-to-left* order refers to searching the triangle from the current node by travelling to the node on its *left*. A *left-to-right* order refers to searching the triangle from the current node by travelling to the node on its *right*.

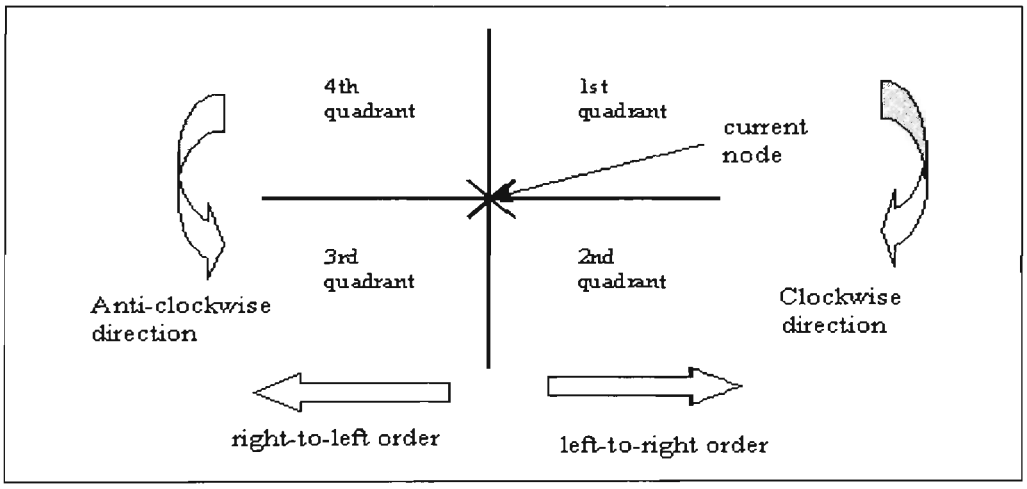

Figure 5.5: Direction of searching for triangle

The procedure to search for the triangle is based on the overall direction of the tour: clockwise or anti-clockwise. Assuming the case where the direction of travel is clockwise, if there exists candidate neighbouring nodes to the left of the current node, then the search of the triangle will be made in a *right-to-left* order from the third quadrant in a clockwise direction. Otherwise, the search will be carried out in a *left-to-right* order starting from the first quadrant in a clockwise direction if the candidate

neighbouring nodes are to its right. In the other case where the direction of travel is anti-clockwise, if there are candidate neighbouring nodes to its right, then the search of the triangle is made in a *left-to-right* order starting from the second quadrant in anti-clockwise direction. Otherwise, the search is carried out in a *right-to-left* order starting from the fourth quadrant in anti-clockwise direction if there are candidate neighbouring nodes to its left.

The following 4-city example shown in Figure 5.6 is used to explain the above idea. If the tour is travelling in a clockwise direction and the tour starts with node 2, then the complete tour will be 2-1-4-3-2. On the other hand, a tour that is travelling in an anti-clockwise direction will visit the nodes in the order of 2-3-4-1-2. In both cases, the tour is the same however the order the cities travel is different. Thus if the tour starts with node 2 and the tour travels in a clockwise direction, then the search of triangle will be performed in a *right-to-left* manner from the third quadrant because there is one node to the left of node 2. This means the search of the next node to be added to node 2 will begin from the third quadrant of node 2 in a clockwise direction. This way it will select neighbouring nodes based on triangle 2-1-4.

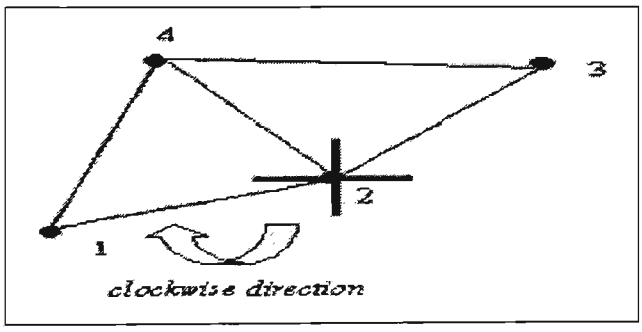

Figure 5.6: Example 1 to illustrate the search direction

It is important to note that it is the overall direction of the tour that decides which way to search for the candidate neighbouring nodes. To illustrate this point, assuming the tour now starts at node 4 and the direction of the tour is clockwise (see Figure 5.7).

There are neighbouring nodes on both sides of node 4, which of the above rule should be applied? In this case, the search should begin from the first quadrant of node 4, and not the third quadrant, in a clockwise direction. If the search were to begin from the third quadrant, then the tour formed is 4-1-2-3-4, which travels in anti-clockwise direction. On the other hand, by starting the search from the first quadrant, then the tour formed is 4-3-2-1-4, which is in a clockwise direction.

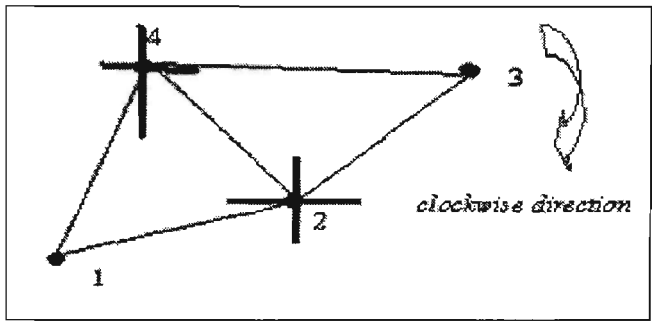

Figure 5.7: Example 2 to illustrate the search direction

Finally, the question still remains as to which triangle from Delaunay triangulation is to be selected during the tour construction process. It is reasonably clear that if possible, one should avoid long edges when searching for optimal tour, because too many long edges in the tour will not lead to shorter tour. Therefore the proximity candidate edge set provides useful information that can be used in identifying the triangle. Convex hull provides a good initial subtour, and internal nodes inside the convex hull can be inserted between the external nodes to form a tour in the order in which the tour travels along the boundary of the convex hull. If there are internal nodes between two external nodes, then it is clear that they will be inserted between the external nodes by following the direction of the tour. By using the characteristic of the convex hull described above, and at the same time, taking the advantage of direction of travels, it is proposed in this research to select the first triangle (by following the search direction as defined above) that has both neighbouring nodes in the proximity

candidate edge set, and both nodes must not have been selected before. If the triangle contains nodes that have been selected before, then the search will continue to the next adjoining triangle until a triangle that satisfies the above condition is found. In the event no triangle is found, then it is required to start the search process again to find the first triangle that has at least one of its neighbouring nodes in the proximity candidate edge set. The procedure to find the triangle is summarised in procedure find_triangle in Figure 5.8.

procedure find_triangle

begin

1). For each node i, find proximity candidate edge set P_i using procedure find_proximity_candidate_edge_set.

2). For each node, divide the node into four quadrants. In a clockwise direction name the quadrants: first, second, third and fourth.

3). Determine the direction of the tour: clockwise or anti-clockwise.

4). If the direction of the tour is clockwise, do rule 1 or 2:

Rule 1
If there are more unvisited candidate neighbouring nodes to its *left*, start the search in a *right-to-left* order from the third quadrant.
Rule 2
If there are more unvisited candidate neighbouring nodes to its *right*, start the search in a *left-to-right* order from the first quadrant.

Go to step (6).

5). If the direction of the tour is anti-clockwise, do rule 3 or 4:

Rule 3
If there are more unvisited candidate neighbouring nodes to its *right*, start the search in a *left-to-right* order from the second quadrant.
Rule 4
If there are more unvisited candidate neighbouring nodes to its *left*, start the search in a *right-to-left* order from the fourth quadrant.

Go to step (6).

6). Let i be the last city added to the tour, follow the search direction defined above to find the first triangle that has both candidate neighbouring nodes in P_i. If not found, then return to start the search process again to find the first triangle that has at least one candidate neighbouring node in P_i.

end

Figure 5.8: Procedure of find_triangle

The procedure to find triangle as described above is simple in concept. However we may face a situation where sub-optimal solution is obtained because a very limited

number of candidate edges are considered during the search process. It is possible that in situation where there are a large number of internal nodes grouped near to one another, the rules as defined in the procedure find_triangle may not be that easy and straightforward to apply. In circumstances like this, it would be advantageous to include more candidate neighbouring nodes from more than one triangle so that all promising edges that are likely to be in the optimal tour are not omitted during the restrictive search process. Therefore, it is suggested that the search for triangle should be augmented to include all triangles either to its left or right depending on the direction the tour travels. This way, the overall direction of the tour still plays an important role in deciding which triangles are to be selected. The rule to search for the triangle may be modified as follows when one wants to augment the search space to include candidate neighbouring nodes from more than one triangle. If the tour travels in a clockwise direction, then start the search for triangles in a left-to-right manner so that all triangles located to its right are selected. Similarly, if the tour travels in an anti-clockwise direction, then start the search in a right-to-left manner so that all triangles to its left are included for consideration. This is a cautious approach to ensure all promising edges that are likely to form the optimal tour are not excluded from consideration during the tour construction process. It is necessary to maintain a balance between reducing the search space and at the same time ensuring that all possible promising neighbouring nodes are included for consideration during the search process.

The search space of the restrictive search approach proposed in this chapter is much smaller comparing to the approach of SLA*-TSP in Chapter 3. By combining the properties associated with the proximity concept of Voronoi diagram, the direction of the tour and the way the triangle is searched, together with the characteristic of optimal

tour and convex hull, the restrictive approach is able to consider only neighbouring nodes that are most likely to result in optimal tour.

5.3 Restrictive SLA*-TSP approach

This section presents the modified procedure of SLA*-TSP approach by incorporating the procedure for finding proximity candidate edge set and triangle as described in Section 5.2. The step-by-step application procedure is the same as that presented in Chapter 3, except steps 0 and 4. Step 0 is replaced with procedure *find_proximity_candidate_edge_set* to find the candidate neighbouring nodes. Step 4 of the algorithm has been modified, by including the procedure find_triangle as part of the search strategy. The modified procedure of the SLA*-TSP that includes the new search strategy as defined above is called the restrictive SLA*-TSP approach, and is given below:

Let S_i be the i^{th} state with its tour $P_i(1,2,...,i,1)$, where 1 is the city of origin and i is the last city of the tour. Its heuristic estimate h(i) is the minimum spanning tree of the remaining (n-i) cities. S_i is the goal state when i = n.

d(i,j) be the Euclidean distance between city i and city j.

H(i) be the estimated tour length for S_i, which consists of the tour P_i and h(i).

Step 0: Apply procedure *find_proximity_candidate_edge_set*.

Step 1: Locate the city of origin as the one with the smallest x-coordinate; choose the city with the largest y-coordinate to break ties.

Step 2: Put the root state on the backtrack list called OPEN.

Step 3: Call the top-most state on the OPEN list S_i. If S_i is the goal state, stop. Otherwise continue.

Step 4: Use the search strategy defined in procedure *find_triangle* to find the $(i+1)^{th}$ city with $\min\{[d(1,2)+d(2,3)+\ldots+d(i-1,i)+d(i,i+1)+d(i+1,1)] +h(i+1)\}$ from neighbouring cities of i; break ties randomly. If no neighbouring city of i can be found, go to step 6.

Step 5: If $\{[d(1,2)+d(2,3)+\ldots+d(i-1,i)+d(i,1)] + h(i)\} \geq \min\{[d(1,2)+d(2,3)+\ldots+d(i-1,i)+d(i,i+1)+d(i+1,1)] + h(i+1)\}$, add S_{i+1} to the OPEN list as the top-most state; otherwise replace h(i) with $[d(i,i+1)+d(i+1,1) + h(i+1) - d(i,1)]$.

Step 6: Remove S_i from OPEN list if it is not the root state.

Step 7: Go to step 3.

5.4 Examples

Three examples are included in this section to demonstrate the implementation of the restricted SLA*-TSP approach. The first example is the 8-city problem, which was given in Chapter 3. The second example is a 14-city problem (BURMA14) that was used in the computation experiment in Chapter 4. The third example is a randomly generated 12-city problem. Finally a discussion in terms of performances with the two approaches: SLA*-TSP and restrictive SLA*-TSP, is presented.

5.4.1 Example 1

Example 1 is the 8-city problem which was used in Chapter 3 for demonstration of SLA*-TSP (see Section 3.4.1, page 37). By using the procedure given in Section 5.2, the first step is to identify the proximity candidate edge set for each city. In this example, the convex hull is made up of cities {1,4,3,6,7}, and Voronoi diagram was constructed based on these five cities (see Figure 5.3, page 77). Using the procedure find_proximity_candidate_edge_set, the proximity candidate edge set for each city is identified. Table 5.1 shows the elements in the candidate edge set derived from Delaunay triangulations, Voronoi diagram using external cities only, and proximity candidate edge set. Column 1 shows the city and column 2 indicates if the city is on convex hull. Column 3 presents the candidate edge set developed using Delaunay triangulation D_i, column 4 gives a list of internal cities that are in the same Voronoi region as the city on convex hull (V_i), and column 5 shows the proximity candidate edge set $P_i = \{D_i \cap V_i\}$. For internal cities that are not on the convex hull, a "-" in column 4 is used to show that V_i is not applicable. Table 5.1 also shows which rule from the procedure find_triangle has been applied, and which triangle has been used to identify the candidate neighbouring cities during the search process.

The complete forward search process for this problem is given in Table 5.2. The presentation of Table 5.2 is the same as the presentation format used in Chapter 3 in which only the forward search processes are shown. The step-by-step implementation of the restrictive SLA*-TSP approach will not be described here. The discussion will focus on how the triangle is selected for each city during tour construction process and the rule that is used during the search process. The discussion will be presented in the order the cities are visited in a clockwise direction.

City i	On convex hull	Candidate edge set		Proximity candidate edge set, P_i	Rule applied (procedure find_triangle)	Triangle selected
		From Delaunay triangulation, D_i	From Voronoi diagram, V_i			
1	√	4, 7	∅	4, 7	Rule 2	△1-7-4
2	×	3, 5, 6, 8	-	3, 5, 6, 8	Rule 1	△2-3-5
3	√	2, 4, 5, 6	2, 5, 8	2, 5	Rule 2	△3-5-2
4	√	1, 3, 5, 7, 8	∅	1, 3, 5, 7, 8	Rule 2	△4-1-7
5	×	2, 3 4, 8	-	2, 3 4, 8	Rule 2	△5-2-8
6	√	2, 3, 7, 8	∅	2, 3, 7, 8	Rule 1	△6-3-2
7	√	1, 4, 6, 8	∅	1, 4, 6, 8	Rule 1	△7-6-8
8	×	2, 4, 5, 6, 7	-	2, 4, 5, 6, 7	Rule 2	△8-6-2

Table 5.1: Proximity candidate edge set for the 8-city problem

Iteration	Level-0	Level-1	Level-2	Level-3	Level-4	Level-5	Level-6	Level-7
1	(4,4) * 13070	(4,1,4) 15891						
2	(4,4)* 15891	(4,1,4) 15891	(4,1,7,4) 15740	(4,1,7,8,4) 18548				
3	(4,4)* 18548	(4,1,4) 18548	(4,1,7,4) 18548	(4,1,7,8,4) 18548	(4,1,7,8,6,4) 24215			
4	(4,4)* 23093	(4,1,4) 23093	(4,1,7,4) 23093	(4,1,7,6,4) 23093	(4,1,7,6,2,4) 22218	(4,1,7,6,2,3,4) 21931	(4,1,7,6,2,3,5,4) 21480	(4,1,7,6,2,3,5,8,4) 22300
5	(4,4) 23093	(4,1,4) 23093	(4,1,7,4) 23093	(4,1,7,6,4) 23093	(4,1,7,6,2,4)* 22300	(4,1,7,6,2,3,4) 22300	(4,1,7,6,2,3,5,4) 22300	(4,1,7,6,2,3,5,8,4) 22300

Table 5.2: Forward search process for 8-city problem using the restrictive approach

The direction of the tour is clockwise, and the starting city is city 4. From city 4, rule 2 of procedure find_triangle is applied. Rule 2 is used in this instance because its unvisited candidate neighbouring cities are located to its right. The first triangle that has both candidate neighbouring cities in the proximity candidate edge set P_4 is triangle △1-4-7. Therefore the selection of the neighbouring city to the tour can be made from city 1 or 7, and city 1 is selected because it has the minimum estimated tour length. From city 1, rule 2 is applied because its unvisited candidate neighbouring city is to its right. The search begins from the first quadrant. However, there is no triangle in the first quadrant and the search continues to the second quadrant. It is found that there is no triangle that satisfies the condition that both unvisited candidate

neighbouring cities in P_1, therefore the search process continues. This time the triangle with at least one unvisited neighbouring city in P_1 is selected, and the triangle is △1-4-7. In this case, the neighbouring city for city 1 is city 7; city 4 is excluded because it is already in the partial tour. From city 7, rule 2 is once again applied because its neighbouring node is to its right. This time the first triangle encountered is △7-6-8 and the state generated is state (4,1,7,8,4). This state has the minimum estimated tour length that is greater than the estimated tour length of its parent state, therefore backtracking occurs and heuristic update takes place (see iteration 2 as shown in Table 5.2).

In the third iteration, the search progresses from (4,4), (4,1,4), (4,1,7,4), (4,1,7,8,4). The last city added to the partial tour is city 8, and rule 2 is applied to find the triangle. Although both sides of city 8 have unvisited neighbouring cities, it is rule 2 that is applied because it will result in a tour that is consistent with the clockwise direction of travel. The first triangle that has both unvisited neighbouring cities is △8-6-2.

In iteration four, the search starts from (4,4), and continues to (4,1,4), (4,1,7,4), (4,1,7,6,4). The last city in this partial tour is city 6, which has its unvisited neighbouring nodes to its left. This time rule 1 is applied, and the first triangle encountered is △2-3-6 and the state with the minimum estimated tour length of 22218 is (4,1,7,6,2,4). From city 2, rule 1 is again applied and the first triangle encountered is △2-3-5. From city 3, rule 2 is used and the first triangle is △3-5-2. From city 5, rule 2 is again used and the triangle found is △5-2-8, and the state generated is (4,1,7,6,2,3,5,8,4). At this stage, more heuristic estimates and backtracking steps will be carried out until an optimal tour of (4,1,7,6,2,3,5,8,4) with tour length 22300 is

found (see iteration 5 in Table 5.2). Results from Table 5.2 shows that the restrictive approach only takes five forward search trials to reach the optimal solution.

5.4.2 Example 2

Example 2 is a problem of fourteen cities (BURMA14) obtained from TSPLIB (Reinelt, 1991). This problem was one of the four problems used in the performance analysis of SLA*-TSP in Chapter 4. Figure 5.9 shows the combined Delaunay triangulations and Voronoi diagram using the approach described in Section 5.2. In the diagram, single-line shows the Delaunay triangulations and the bold-line shows the Voronoi regions.

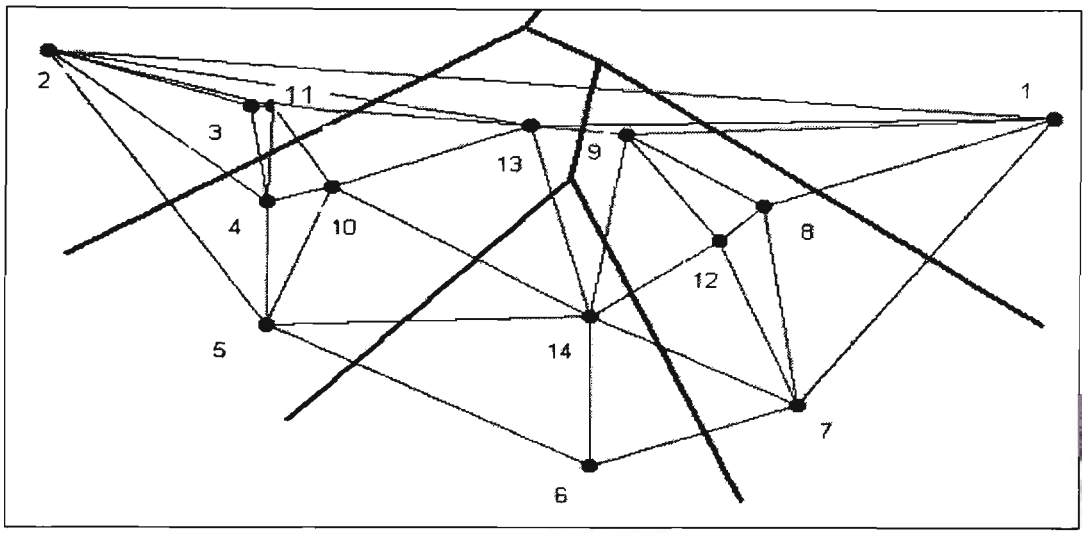

Figure 5.9: Combined Delaunay triangulations and Voronoi diagram for BURMA14

Table 5.3 shows the elements in the candidate edge sets and the triangle that is found for each city. The layout and presentation format of Table 5.3 are the same as Table 5.1 above. In this example, it can be seen that proximity candidate edge sets for internal cities are the same as the candidate edge set defined using Delaunay triangulations, however the proximity candidate edge sets for external cities 2, 5, 6, 7 have reduced, except city 1 which remains the same.

As with the previous example, the explanation on how the forward search and backtracking processes took place in the restrictive SLA*-TSP approach will not be discussed. On the other hand, the discussion concentrates on examining how the triangles are determined and the rule that has been used during the search process. The discussion will be presented in the order the cities are visited in a clockwise direction. The direction the tour travels is clockwise, and the tour starts from city 5.

City i	On convex hull	Candidate edge set		Proximity candidate edge set, P_i	Rule applied (procedure find_triangle)	Triangle selected
		From Delaunay triangulation, D_i	From Voronoi diagram, V_i			
1	√	2,7,8,9,13	∅	2,7,8,9,13	Rule 1	△1-7-8
2	√	1,3,4,5,11,13	3,11	3,11	Rule 2	△2-3-11
3	×	2,4,11	-	2,4,11	Rule 2	△3-2-11
4	×	2,5,10,11	-	2,5,10,11	Rule 1	△4-2-3
5	√	2,4,6,10,14	4,10,13	4,10	Rule 1	△5-4-10
6	√	5,7,14	14	14	Rule 1	△6-5-14
7	√	1,6,8,12,14	8,9,12	8,12	Rule 1	△7-6-14
8	×	1,7,9,12	-	1,7,9,12	Rule 2	△8-1-7
9	×	1,8,12,13,14	-	1,8,12,13,14	Rule 2	△9-1-8 △9-8-12
10	×	4,5,11,13,14	-	4,5,11,13,14	Rule 2	△10-13-14
11	×	2,3,4,10,13	-	2,3,4,10,13	Rule 2	△11-10-13
12	×	7,8,9,14	-	7,8,9,14	Rule 2	△12-8-7
13	×	1,2,9,11,14	-	1,2,9,11,14	Rule 2	△13-9-1
14	×	5,6,7,9,10,12,13	-	5,6,7,9,10,12,13	-	-

Table 5.3: Proximity candidate edge set for BURMA14

The tour starts with city 5, and it is likely to travel to the neighbouring nodes to its left; otherwise it will not lead to a tour that travels in clockwise direction. Therefore rule 1 of procedure find_triangle is used, and the first triangle that has both candidate neighbouring cities is △5-4-10. Using the algorithm of SLA*-TSP, city 4 is included in the partial tour. From city 4, rule 1 is again applied and the triangle selected is △4-

2-3. This time, city 2 is added to the partial tour. From city 2, rule 2 is applied because its unvisited neighbouring cities are to its right. The proximity candidate edge set for city 2, P_2 = {3, 11}, therefore the first triangle that has both unvisited candidate neighbouring cities in P_2 is △2-3-11. From city 3, rule 2 is again applied and the triangle selected is △3-2-11. In this instance, there is no triangle that has both unvisited neighbouring cities in proximity candidate edge set P_3. Therefore the search needs to start again, and the triangle that has at least one unvisited neighbouring city is △3-2-11. The same rule (i.e. rule 2) applies for cities 11, 10, 13, 9, 12, and 8, because each of these cities has unvisited neighbouring nodes located to its right. In the case of node 9, the cautious approach that has been discussed in Section 5.2 has been used. Instead of considering candidate neighbouring nodes from only one triangle, all triangles that are to its right are considered. In this instance, triangles △9-8-12 and △9-1-8 have been selected. After that, rule 1 is applied to cities 1, 7 and 6, because it must travel to the neighbouring nodes to its left in order for the tour to travel in a clockwise direction. The last city is city 14, and it is not necessary to select any triangle for this city. The results are summarised in Table 5.3.

The complete forward search trials are shown in Table B.1 in Appendix B. The results from Table B.1 shows that it takes 35 forward search trials to reach the optimal solution using the restrictive SLA*-TSP approach.

5.4.3 Example 3

The test approach in this example is adapted from the randomly generated problem in Chapter 4. It is a 12-city randomly generated problem that exhibits clustering of nodes. It consists of four clusters with 3 nodes in each cluster. The nodes are located within a (0,100) square and the square is divided into 16 equal rectangles (see Figure 5.10).

Three nodes are randomly generated within each rectangle according to a uniform distribution. In this example, the nodes are selected from square 1, 4, 13 and 16.

4	8	12	16
3	7	11	15
2	6	10	14
1	5	9	13

Figure 5.10: Structure of problems tested

Figure 5.11 shows the combined Delaunay triangulations and Voronoi diagram of the problem. As with previous examples, the Delaunay triangulations are constructed based on all nodes in the problem and the Voronoi diagrams constructed are based on external nodes only. The single-line shows the Delaunay triangulation and the bold lines show the boundary of Voronoi regions formed by nodes that lie on the convex hull. Table 5.4 shows the elements in the candidate edge set D_i, V_i and P_i. The layout and presentation format of Table 5.4 is the same as Table 5.1

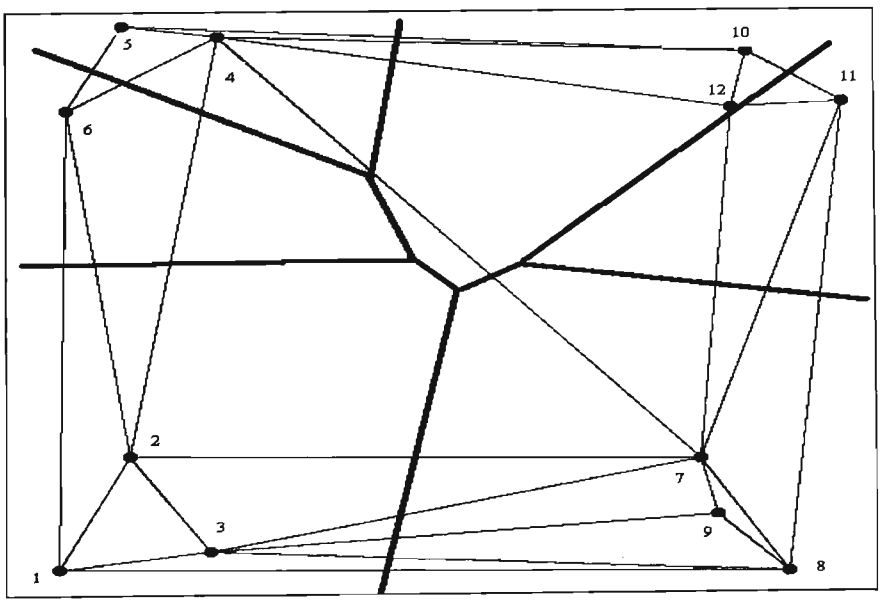

Figure 5.11: The combined Delaunay triangulation and Voronoi diagram of Example 3

City i	On convex hull	Candidate edge set		Proximity candidate edge set, P_i	Rule applied (procedure find_triangle)	Triangle selected
		From Delaunay triangulation, D_i	From Voronoi diagram, V_i			
1	√	2,3,6,8	2,3	2,3	Rule 3	△1-8-3
2	×	1,3,4,6,7	-	1,3,4,6,7	Rule 4	△2-1-3
3	×	1,2,7,8,9	-	1,2,7,8,9	-	-
4	×	2,5,6,7,10,12	-	2,5,6,7,10,12	Rule 4	△4-5-6
5	√	4,6,10	4	4	Rule 4	△5-4-6
6	√	1,2,4,5	∅	1,2,4,5	Rule 4	△6-1-2
7	×	2,3,4,8,9,11,12	-	2,3,4,8,9,11,12	Rule 3	△7-11-12
8	√	1,3,7,9,11	7,9	7,9	Rule 4	△8-7-9
9	×	3,7,8	-	3,7,8	Rule 4	△9-7-3
10	√	4,5,11,12	12	12	Rule 4	△10-4-5
11	√	7,8,10,12	∅	7,8,10,12	Rule 4	△11-10-12
12	×	4,7,10,11	-	4,7,10,11	Rule 3	△12-10-4

Table 5.4: Proximity candidate edge set of Example 3

As with Example 2, this section focuses on how the triangle is selected and the application of SLA*-TSP will not be discussed. It is assumed that the tour travels in an anti-clockwise direction, and the tour starts with city 8. From city 8, rule 4 of procedure find_triangle is applied, because its neighbouring nodes are to its left. The first triangle that has both candidate neighbouring nodes in the proximity candidate edge set of P_8 is △8-7-9. The next city to consider is city 9. Again rule 4 is applied because its neighbouring cities are to its left. This time △9-7-3 is selected. This is followed by city 7, and rule 3 is used to find triangle △7-11-12. City 11 uses rule 4 to find triangle △11-10-12 because its neighbouring cities are to its left. Then city 12 uses rule 3 to find △12-10-4. Rule 4 is applied to cities 10, 4, 5, 6, 2, and triangles found are △10-4-5, △4-5-6, △5-4-6, △6-1-2, and △2-1-3 respectively. Then city 1 uses rule 3 to find triangle △1-8-3 and city 3 is the last city added to the tour. The results is summarised in Table 5.4.

Table B.2 of Appendix B shows the complete forward search trials of this problem. From Table B.2, it shows that only 14 forward search trials are required to obtain optimal solution using the restrictive search approach.

5.4.4 Discussion

Table 5.5 summarises the results for the above three examples. The table compares the number of forward searches each problem takes using both the SLA*-TSP and the restrictive search approach. The number of heuristic updates needed in both approaches for each problem is also included in the table. To compare the performance of these two approaches, the ratio of $\frac{RestrictiveSLA*-TSP}{SLA*-TSP}$ is computed for the number of forward search as well as that of heuristic updates.

		Number of forward search			Number of heuristic updates		
Example	No of cities	SLA*-TSP	Restrictive SLA*-TSP	Ratio	SLA*-TSP	Restrictive SLA*-TSP	Ratio
1	8	18	5	28%	74	9	12%
2	14	786	33	4%	5601	182	3%
3	12	26	14	54%	124	37	30%

Table 5.5: Summary of results for three examples

Results from Table 5.5 show that the restrictive SLA*-TSP approach substantially outperforms the SLA*-TSP approach. Problem in Example 2 shows that the number of heuristic updates in the restrictive search approach is only 3% of the SLA*-TSP approach, with a saving of 97%. Even the clustered problem in Example 3, which is the worst of the three, the number of heuristic updates in the restrictive SLA*-TSP approach is only 30% of the SLA*-TSP approach, with a saving of 70%. As discussed in Chapter 4, in problems that exhibit clustering or grouping of nodes, the value of the

heuristic estimate is greater than otherwise. This is because the minimal spanning tree includes those edges that are between the clusters and results in a higher value of heuristic estimate. Therefore problems that exhibit clustering or grouping of nodes will be relatively advantageous to the approach of SLA*-TSP. On the other hand, when the restrictive search approach is applied to problems that do not exhibit clustering characteristics, a very substantial saving in terms of the number of heuristic updates can be achieved.

Table 5.6 shows the results of the number of states generated in the search process. The number of states is calculated based on the number of candidate neighbouring nodes generated. It can be seen that the restrictive search approach generates fewer states. This is because each time the tour is expanded, it considers a very limited number of promising neighbouring nodes depending on the direction of the tour. On the other hand, in the SLA*-TSP approach it needs to consider all unvisited neighbouring nodes that are connected through the edges of Delaunay triangulation.

Example	No of cities	SLA*-TSP	Restrictive SLA*-TSP	Ratio
1	8	90	16	18%
2	14	34560	512	2%
3	12	4800	512	11%

Table 5.6: Number of states generated in the search process

Initial study from these three examples has indicated that the search space of SLA*-TSP approach can be much improved by considering only the most promising candidate neighbouring nodes during the tour construction process. The improvement is achieved through the use of knowledge pertaining to the convex hull, Voronoi diagram, direction of the tour and the direction of search.

5.5 Conclusion

In this chapter, it has shown that knowledge based on the computational geometric characteristics of Euclidean TSP can be utilised to identify a reduced set of candidate nodes to be included in the search process. The contribution of this chapter is the development of the restrictive SLA*-TSP approach in which the search strategy has utilised the characteristics of Voronoi diagram, direction of the tour, and the direction of search. In this restrictive approach, only the most promising neighbouring nodes are included for consideration during the tour construction process. It is believed that the restrictive SLA*-TSP approach could present an effective alternate approach in addressing TSP; in particular the selection of promising cities based on the proximity information of Voronoi diagram and the way the direction is integrated in the search strategy.

CHAPTER 6: CONCLUSIONS

6.1 Overview of research

The principal emphasis of this research is the development of a heuristic learning approach to construct TSP tour by utilising the underlying computational geometric properties of Euclidean TSP. This thesis centred on two issues of tour construction heuristics that had not been pursued previously.

1. Greedy and myopic natures of tour constructions.

   This thesis developed a dynamic tour construction approach by using the heuristic learning approach of SLA*. Dynamic tour construction aims to change the configuration of the tour while the tour is under construction. The backtracking and heuristic updating features in SLA* offer an opportunity for this to occur.

2. Search strategy based on the underlying computational geometric properties of Euclidean TSP .

   This thesis focuses on two-dimensional Euclidean TSP. Computational geometric properties associated with Euclidean TSP such as Delaunay triangulation and Voronoi diagrams are utilised to ensure that only promising cities that are likely to lead to an optimal tour are included for consideration in the tour construction process.

This research shows that SLA*-TSP is a powerful heuristic learning approach that can be used to construct tour dynamically. The tour configuration process is constantly changing during the tour construction process until a solution is obtained. It has demonstrated that by utilising the computational properties and characteristics associated with Euclidean TSP, the search process considers only promising cities that are likely to result in optimal solution. Knowledge concerning Delaunay triangulation, Voronoi diagram, the direction of the tour travels and direction of search for triangle can be embedded in a restrictive search approach to reduce search space.

6.2 Results of research

In the pursuit of the above topics, the following results were obtained.

1. Development of SLA*-TSP approach.

   A state space transformation process that includes state definition, state transition operator and state transition cost has been developed. This is to allow TSP to be formulated as a state-space problem so that SLA*-TSP can be applied. The implementation approach of SLA*-TSP has been developed.

2. Development of two search strategies based on the underlying computational geometric properties of Euclidean TSP.

   The first strategy utilises the characteristics of Delaunay triangulations in defining search space. The second strategy is more restrictive, it utilises the proximity property of Voronoi diagram, convex hull, direction the tour travels and the direction of search for an appropriate triangle. This latter strategy

selects an appropriate triangle from Delaunay triangulation, which contains information of the candidate edges that are likely to lead to the optimal tour.

3. Investigation of quality of heuristic estimate in determining the performance of SLA*-TSP.

   The distribution of nodes on the problem influences the heuristic estimate because it influences the computation of minimal spanning tree. In problems that exhibit clustering or grouping of nodes, it will lead to a better quality of heuristic estimate. This is because when the clusters are well separated, long edges linking clusters tend to be included in the minimal spanning tree.

The research also shows that the application of SLA*-TSP with a learning threshold approach can be used to improve the computation time for large sized problems at the sacrifice of the optimal solution. By including the learning threshold in the algorithm, SLA*-TSP with learning threshold approach is able to find an approximate solution with a known quality. The advantage of this approach is that the maximum amount of sacrifice is known before hand.

6.3 Contribution of the research

Conventional tour construction heuristics is myopic and greedy in nature. This research addresses the above gaps in conventional tour construction heuristics for traveling salesman problem. With the application of heuristic learning algorithm SLA*, the work presented here shows that the tour can be constructed dynamically through the use of local and global distance information. Along the search process, the backtracking and forward search processes can repetitively lead to deletion and

addition of cities from and to the tour. This dynamic construction of the tour is made possible through the consideration of both the local tour information and the global tour estimation, which is updated continuously along the search process.

The work presented here shows that by utilising the computational geometric properties of Euclidean TSP, the performance of the heuristic learning feature of SLA*-TSP can be enhanced. Firstly, the search capability of SLA*-TSP is enhanced by the solution space selection of the Delaunay triangulations. In addition, through the integration of Voronoi diagram, operating as a decision boundary, with the direction of search for the triangle, a more intelligent restrictive SLA*-TSP approach has been developed. This restrictive approach makes the selection of promising cities based on the proximity information of Voronoi diagram, and the sense of direction of search is integrated in the search strategy to avoid unnecessary search. This is an approach that has not been pursued previously.

This research has demonstrated that a well-developed heuristic learning algorithm in artificial intelligence such as SLA* can present an effective alternate way of addressing traditional TSP. The research has shown that the heuristic learning approach together with the underlying computational geometric properties of Euclidean TSP can be integrated in such a way that the tour can be constructed dynamically in which the configuration of the tour can be constantly updated during the tour construction process. The characteristics of computational geometry has been utilised to define a reduced search space so that only the promising candidate edges are included in the tour construction process.

6.4 Future research

Based on the findings of this research, future research needs to be conducted to offer a more complete and comprehensive heuristic learning framework to solve TSP. The following issues warrant further investigation.

- One of the major weaknesses of the current computer implementation is its inefficient memory handling. Further research needs to be carried out to improve the design of the data structures to improve the efficiency of memory handling. One of the possible methods is to use tree structure so that the state and the value of the updated heuristic estimate can be stored and retrieved more efficiently.

- The major factor influencing the performance of SLA*-TSP is the heuristic estimate. Minimal spanning tree is used in this research to compute the heuristic estimate. However, further research can to be investigated to find an alternative method, if any, to compute the heuristic estimate that has a value as close to the optimal solution as possible. Investigations can be conducted to determine the suitability of other lower bound, such as assignment problem, to be used as a heuristic estimate.

- The SLA*-TSP with learning threshold approach allows the algorithm to find an approximate solution with a desired range of certainty. This feature is particularly important in practical situations when an exact solution is not required, but the speed of computation is. However, there is a need to better understand the way backtracking is delayed in relation to the neighbourhood

and geometric properties of the nodes when learning threshold is included. Further research needs to be conducted to examine this relationship.

REFERENCES

Angeniol, B., Vaubois, G. and Texier, J. 1988. Self-Organising Feature Maps and the Traveling Salesman Problem. *Neural Networks*, 1, 289-293

Applegate, D., Bixby, R., Chvátal, V. and Cook, W., 1998, On the Solution of Traveling Salesman Problems, *Doc.Math.J.DMV* Extra Volume ICM III 645-656

Aurenhammer, F. 1991. Voronoi Diagrams – A Survey of a Fundamental Geometric Data Structure. *ACM Computing Surveys*, 23(3), 345-405

Baker, K. 1974. *Introduction to Sequencing and Schedu*ling. John Wiley & Sons

Bartholdi, J. and Platzman, L. 1988. Heuristics Based on Spacefilling Curves for Combinatorial Problems in Euclidean Space. *Management Science*, 34, 291-305

Bentley, J.J. 1992. Fast Algorithms for Geometric Traveling Salesman Problems. *ORSA Journal on Computing*, 4(4), 387-411

Burke, L. 1994. Neural Methods for the Traveling Salesman Problem: Insights from Operations Research. *Neural Networks*, 7(4), 681-690

Carpaneto, G. and Toth, P. 1980. Some New Branching and Bounding Criteria for the Asymmetric Traveling Salesman Problem. *Management Science*, 26, 736-743

Carpaneto, G., Dell'Amico, M. and Toth, P. 1995. Exact Solution of Large-Scale Asymmetric Traveling Salesman Problems. *ACM Transactions on Mathematical Software*, 21(4), 394-409

Cerny, V. 1985. A Thermodynamical Approach to the Traveling Salesman Problem: an Efficient Simulation Algorithm. *Journal on Optimization Theory and Application*, 45, 42-51

Cesari, G. 1996. Divide and Conquer Strategies for Parallel TSP Heuristics. *Computers Operations Research*, 23(7), 681-694

Christofides, N. and Eilon, S. 1972. Algorithms for Large-Scale Traveling Salesman Problems. *Operational Research Quarterly*, 23(4), 511-518

de Berg, M., van Kreveld, M, Overmars, M. and Schwarzkopf, O. 1997. *Computational Geometry: Algorithms and Applications*, Springer-Verlag, Berlin

Dowsland, K. 1995. Simulated Annealing, in Reeves, C. (ed) *Modern Heuristic Techniques for Combinatorial Problems*, Haksted Press, 20-69

Garey N.R. and Johnson, D.S. 1979. *Computers and Intractability: a Guide to the Theory Of NP-Completeness*. W.H Freeman

Geman, S. and Geman, D. 1984. Stochastic Relaxation, Gibbs Distributions and The Bayesian Restoration of Images. *IEEE Trans on Pattern Analysis and Machine Intelligence*, PAMI-6, 721-741

Gendreau, M., Hertz, A. and Laporte, G. 1992. New Insertion and Postoptimization Procedures for the Traveling Salesman Problem. *Operations Research*, 40(6), 1086-1094

Gendron, B. and Crainic, T.G. 1994. Parallel Branch-and-Bound Algorithms: Survey and Synthesis. *Operations Research*, 42 (6), 1042-1066

Glover, F. 1990. Artificial Intelligence, Heuristic Frameworks and Tabu Search. *Managerial and Decision Economics*, 11, 365-375

Glover, F. and Laguna, M. 1993. Tabu Search, in Reeves, C. (ed), *Modern Heuristic Techniques for Combinatorial Problems*. John Wiley & Sons

Golden, B., Bodin, L., Doyle, T. and Stewart Jr, W. 1980. Approximate Traveling Salesman Algorithms. *Operations Research*, 28(3), 694-709

Golden, B. and Stewart, W. 1985. Empirical Analysis of Heuristics, in Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G. and Shmoys, D.B. (eds), *The Traveling Salesman Problem*, 207-250, John Wiley & Sons

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, 1994

Holland, J. 1975. *Adaptation in Natural and Artificial Systems*. University of Michigan Press

Hopfield, J.J. and Tank D.W. 1985. Neural Computation of Decisions in Optimisation Problems. *Biol. Cybern.*, 52, 141-152

Jedrzejek, C. and Cieplinski, L. 1995. Heuristic versus Statistical Physics Approach to Optimization Problems. *ACTA Physical Polonica B*, 26(6), 977-996

Johnson, D.S. 1990. Local Optimization and the Traveling Salesman Problem. in *Proceedings 17th Colloquium on Automata, Languages and Programming*, Lecture Note in Computer Science, 443, 446-461

Johnson, D.S. and Papadimitriou, C.H. 1985. Performance Guarantees for Heuristics. in Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G. and Shmoys, D.B. (eds), *The Traveling Salesman Problem*, 145-180, John Wiley & Sons

Karp. R.M. 1979. A Patching Algorithm for the Nonsymmetric Traveling Salesman Problem. *SIAM J. Comput*, 8, 561-573

Kirkpatrick, S., Gelatt, C.D. and Vecchi, M.P. 1983. Optimisation by Simulated Annealing. *Science*, 220, 671-680

Knox, J. 1994. Tabu Search Performance on the Symmetric Traveling Salesman Problem. *Computers Ops. Res.* 21(8), 867-876

Kohonen, T. 1984. *Self-Organisation and Associative Memory*. Springer-Verlag, Berlin

Kolen, A. and Pesch, E. 1994. Genetic Local Search in Combinatorial Optimization. *Discrete Applied Mathematics*, 48, 273-284

Koulamas, C., Anthony, S.R. and Jaen, R. 1994. A Survey of Simulated Annealing Applications to Operations Research Problems. *Omega International Journal Management Science*, 22(1), 41-56

Krasnogor, N., Moscato, P. and Norman, M.G. 1995. A New Hybrid Heuristic for Large Geometric Traveling Salesman Problems Based on the Delaunay Triangulation. *Anales del XXVII Simposio Brasileiro de Pesquisa Operacional*, Vitoria, Brazil, 6-8 Nov 1995

Kruskal, J. 1956. On the Shortest Spanning Subtree of a Graph and the Travelling Salesman Problem. *Proceedings of the American Mathematical Society*, 48-50

Laporte, G., Potvin, J. and Quilleret, F. 1996. A Tabu Search Heuristic using Genetic Diversification for the Clustered Traveling Salesman Problem. *Journal of Heuristics*, 2, 187-200

Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G. and Shmoys, D.B. 1985. *The Traveling Salesman Problem*. John Wiley & Sons

Lin, S. 1965. Computer Solutions of the Traveling Salesman Problem. *Bell Systems Tech. J.* 44, 2245-2269

Lin, S.C. and Hsueh, J.H.C. 1994. Simulated Annealing for the Optimisation Problems. *Physica A*, 205, 367-374

Lin, S. and Kernighan, B.W. 1973. An Effective Heuristic for the Traveling Salesman Problem. *Operations Research*, 21(2), 498-516

Lourenco, H. R. 1995. Job-Shop Scheduling: Computational Study of Local Search and Large-Step Optimisation Methods. *European Journal Of Operational Research*, 83, 347-364

Mak, K.T. and Morton, A.J. 1993. A Modified Lin-Kernighan Traveling Salesman Heuristic. *Operation Research Letters*, 13, 127-132

Mehlhorn, K. and Näher, S. 1999. *LEDA: A Platform For Combinatorial and Geometric Computing*, Cambridge University Press

Miller, D.L. and Pekny, J.F. 1991. Exact Solution of Large Asymmetric Traveling Salesman Problems. *Science*, February, 754-761

O'Rourke, J. 1998. *Computational Geometry in C* (second edition), Cambridge University Press

Padberg, M. and Rinaldi, G. 1991. A Branch and Cut Algorithm for the Resolution of Large-Scale Symmetric Traveling Salesman Problems. *SIAM Review* 33, 60-100

Papadimitriou, C. 1992. The Complexity of the Lin-Kernighan Heuristic for the Traveling Salesman Problem. *SIAM J. Comput.* 21(3), 450-465

Papadimitriou, C. and Steiglitz, K. 1982. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall

Phan, C. 2000. Investigating Delaunay Triangulation as a Basis for a Travelling Salesman Problem Algorithm. *Transactions Of The Oklahoma Junior Academy Of Science* (Http://Oas.Okstate.Edu/Ojas/Phan.Htm)

Preparata, F.P. and Shamos, M.I. 1985. *Computational Geometry: an Introduction.* Springer-Verlag, New York

Press W.H., Tuekolsky, S.A, Vetterling, W. and Flannery, B.P. 1992. *Numerical Recipes in C: the Art of Scientific Computing* (2nd edition). Cambridge University Press

Prim, R. 1957. Shortest Connection Networks and Some Generalizations. *Bell Syst. Tech. J.*, 36, 1389-1401

Reinelt, G. 1991. TSPLIB – A Traveling Salesman Problem Library. *ORSA Journal On Computing* 3(4), 376-384

Reinelt, G. 1992. Fast Heuristics for Large Geometric Traveling Salesman Problems. *ORSA Journal On Computing* 4(2), 206-217

Reinelt, G. 1994. *The Traveling Salesman: Computational Solutions for TSP Applications*. Springer-Verlag

Rosenkrantz, D.J., Stearns, R.E. and Lewis, P.M. 1977. An Analysis of Several Heuristics for the Traveling Salesman Problem. *SIAM J. Comput.* 6(3), 563-581

Stewart Jr., W.R. 1992. Euclidean Traveling Salesman Problems and Voronoi Diagrams, presented at *the ORSA-CSTS Conference*, Williamsburg, VA

Tate, D.M., Tunasar, C. and Smith, A.E. 1994. Genetically Improved Presequences for Euclidean Traveling Salesman Problems. *Mathematical and Computer Modeling*, 20(2), 135-143

Usami, Y. and Kitaoka, M. 1997. Traveling Salesman Problem and Statistical Physics, *International Journal Of Modern Physics B*, 11(13), 1519-1544

Viswanathan, K.V. and Bagchi, A. 1993. Best-First Search Methods for Constrain Two-Dimensional Cutting Stock Problems. *Operations Research*, 41(4), 768-775

Xu, J. and Kelly, J.P. 1996. A Network Flow-Based Tabu Search Heuristic for the Vehicle Routing Problem. *Transportation Science*, 30(4), 379-393

Yip, P. and Pao, Y. 1995. Combinatorial Optimizations with Use of Guided Evolutionary Simulated Annealing. *IEEE Transactions On Neural Networks*, 6(2), 290-295

Yu, D.H. and Jia, J. 1993. A New Neural Network Algorithm with the Orthogonal Optimized Parameters to Solve the Optimal Problems. *IEICE Transactions Fundamentals*, E76A(9), 1520-1526

Zamani, M.R. 1995. *Intelligent Graph-Search Techniques: an Application to Project Scheduling Under Multiple Resource Constraints*. Phd Thesis

Zhang, W. 1999. *State-Space Search: Algorithms, Complexity, Extensions and Applications*, Springer-Verlag, New York

APPENDIX A: PROGRAM LISTING

```
//C++ program using LEDA library for SLA-TSP
//last modified: 17 April 2001

#define PRUNE
#define NODE_TABLES
// commented out FAST_CODE to collect statistics during run time
//#define FAST_CODE


#include <stdio.h>
#include <math.h>
#include <time.h>
#include <values.h>
#include <stdlib.h>
#include <string.h>
#ifndef __BORLANDC__
#include <sys/times.h>
#endif


#include <LEDA/geo_alg.h>
#include <LEDA/graphwin.h>
#include <LEDA/window.h>
#include <LEDA/rat_kernel_names.h>
#include <LEDA/set.h>

#define COLOR_NORMAL          black
#define COLOR_CURRENT_PATH    red
#define COLOR_MIN_PATH        yellow
#define COLOR_DELETED_PATH    green
#define COLOR_PROPOSED_PATH   cyan


#ifndef __BORLANDC__
#define max(a, b)               ((a) < (b) ? (b) : (a))
#define min(a, b)               ((a) > (b) ? (b) : (a))
#endif

#define BIN_SIZE  1000

#ifdef NODE_TABLES
struct heuristicNode
{
      float heuristic;
    float *graphPtr;
};

struct binNode
{
      struct heuristicNode *heuristicNodePtr;
    int count;
    struct binNode *nextBin;
};

#else

struct binNode
{
      struct binNode *left;
      struct binNode *right;
      float heuristic;
```

```
    float *graphPtr;
};

#endif

struct heuristicLookUpTable
{
      struct binNode *binNodePtr;
};

char string1[200];

int numpoints ;
int bin_size = BIN_SIZE;

#ifndef FAST_CODE
bool stepFlag = false;           // -s
bool graphicFlag = false;        // -g
bool statisticsFlag = false;     // -S
bool displayOutputFlag = false;  // -o
bool autoFlag = false;
bool endOfRunGraph = false;
bool graphSelectedFlag = false;
bool atRootFlag = true;
#endif


bool altThresholdFlag = false;
bool thresholdSelectedFlag = false;
bool randomPointFlag = false;    // -r
bool filePointFlag = false;      // -f



float maxX,maxY,minX,minY;   // windows co-ordinate boundaries

float initThreshold = 0.0;          // -t
float currentThreshold = 0.0;

int startPoint = 1;             // -p
char *progname;
bool backtrackFlag = false;


#ifndef FAST_CODE
int  delayGap = 1;
char *outputFile = NULL;        // -O

int stepBut,                  // button variables
      clearBut,
    autoBut,
    exitBut;

unsigned int statsForwardSequenceCount = 0;
unsigned int statsFowardCount = 0;
bool statsforwardFlag = false;

unsigned int statsBacktrackSequenceCount = 0;
unsigned int statsBacktrackSequenceRootCount = 0;
unsigned int statsBacktrackCount = 0;
unsigned int statsBacktrackRoot2Root = 0;
```

```
unsigned int remainingGraphHeuristicUpdates = 0;
unsigned int remainingGraphNoHeuristicUpdates = 0;
unsigned int subGraphHeuristicUpdates = 0;
unsigned int subGraphNoHeuristicUpdates = 0;

window WDD("DELAUNAY TRIANGULATION");
#endif

list<point> L;                          // list of points that make up
the graph
list<point> LPoint;                     // list of nodes removed from the
graph

#ifdef PRUNE
list<int> connectList;                  // used for determine is all no
visited points are still connects, used by path pruning algorthm
int edgeCounter;                        // used by the path pruning
algorithm
set<point> externalPoints;              // a set of all points that are
on the external edges of the graph
set<string> externalEdges;
#endif

GRAPH<point,int> DT;                    // The Graph defining the
Delaunay Triangle
GRAPH<point,int> remainingGraph;        // The Remaining Graph after the
current processing points have been removed

GRAPH<point,int> previousGraph;
GRAPH<point,int> tempPreviousGraph;

node firstNode,                         // first node in the graph that
is to be processed
       currentNode;                       // The current node of the
graph being processed


stack<GRAPH<point,int> > graphStack;


struct heuristicLookUpTable *graphLoopUpTablePtr;
struct heuristicNode *tempGraphHeuristicNodePtr;

struct heuristicLookUpTable *subGraphLoopUpTablePtr;
struct heuristicNode *tempSubGraphHeuristicNodePtr;


//*****************************************************************
***********
node GetFirstNode(list<point>& L,GRAPH<point,int>& DT, int
startPoint);
float GetDistance(point &pStart,point &pEnd);
void deleteNode(point& p,GRAPH<point,int>& G);
int MSTcmp(const edge &e1,const edge &e2);
float calculateMSTPathLength(GRAPH<point,int>& G, list<edge>& el);

#ifndef FAST_CODE
void processMouse(void);
void FindScalingFactors(float *xFact,float *yFact);
void displayMinSpanningPath(GRAPH<point,int>& G,list<edge>& el);
```

```
void displayGraph(GRAPH<point,int>& G, bool clearFlag);
void displayWholeGraph(void );
void displayCurrentPath(list<point>& listPoints);
void  displayProposedPath(point& p1,point& p2);
#endif

int isDeletedPoint(point &pEnd);
node getGraphNodeFromPoint(GRAPH<point,int>& G,point &p1);
void InitLookUpTables(void);

//**********************
#ifdef NODE_TABLES
void StoreGraphHeuristic(GRAPH<point,int>& G,float heuristic,point
&p);
void StoreGraphUpdatedHeuristic(struct binNode *binNodePtr,float
heuristic,float *ptsBuffer);
void InsertGraphUpdatedHeuristic(struct binNode *binNodePtr,float
heuristic,float *ptsBuffer);
int myGraphBsearch(float *ptsBuffer, struct binNode *binNodePtr,int
size);
float getGraphUpdatedHeuristic(GRAPH<point,int>& G,point &p);
//**********************
void StoreSubGraphHeuristic(GRAPH<point,int>& G,point &p,float
heuristic);
void StoreSubGraphHeuristic_2(GRAPH<point,int>& G,point&
endPoint,float heuristic);
void StoreSubGraphUpdatedHeuristic(struct binNode *binNodePtr,float
heuristic,float *ptsBuffer);
void InsertSubGraphUpdatedHeuristic(struct binNode *binNodePtr,float
heuristic, float *ptsBuffer);
int mySubGraphBsearch(float *ptsBuffer, struct binNode
*binNodePtr,int size);
float getSubGraphUpdatedHeuristic(GRAPH<point,int>& G,point &p);
#else
void StoreGraphHeuristic(GRAPH<point,int>& G,float heuristic,point
&p);
void StoreGraphUpdatedHeuristic(struct binNode *binNodePtr,float
heuristic,float *ptsBuffer);
float getGraphUpdatedHeuristic(GRAPH<point,int>& G, point&
startPoint);
//**********************
void StoreSubGraphHeuristic(GRAPH<point,int>& G,point &p,float
heuristic);
void StoreSubGraphHeuristic_2(GRAPH<point,int>& G,point &p,float
heuristic);
void StoreSubGraphUpdatedHeuristic(struct binNode *binNodePtr,float
heuristic,float *ptsBuffer);
float getSubGraphUpdatedHeuristic(GRAPH<point,int>& G, point&
endPoint);
#endif

float redLineLength(void);
int cmpPts(float *p1, float *p2, int count);

void processArguments(int argc,char *argv[]);
void readInGraph(char *fileName);
void generateRandomGraph(int points,int xlow,int ylow,int xhigh,int
yhigh,int prec);
void displayUsage(void);
```

```
//*********************************************************************
***********


int main(int argc, char *argv[])
{

      progname = "sk4";
      processArguments(argc,argv);

#ifndef FAST_CODE
    ofstream progOutput;
    if (outputFile)
    {
            progOutput.open(outputFile,ios::out);
        if (!progOutput.good())
            {
                  fprintf(stderr,"Unable to open output file\n");
                  displayUsage();
                  exit(0);
      }
    }
#endif

      time_t startTime = time(NULL);



#ifndef FAST_CODE
      float xfact,
        yfact;

      if (graphicFlag)
    {
          stepBut = WDD.button("STEP");
      clearBut = WDD.button("CLEAR");
            autoBut = WDD.button("AUTO");
          exitBut = WDD.button("EXIT");

            FindScalingFactors(&xfact,&yfact);
      WDD.init(minX-xfact,maxX+xfact,minY-yfact);
          WDD.display(window::center,window::center);
      WDD.start_buffering();
    }
#endif

      InitLookUpTables();

// setup DELAUNAY_TRIANGLE DIAGRAM using list L in graph DT
    DELAUNAY_TRIANG(L,DT);

#ifndef FAST_CODE
      if (graphicFlag)
      displayGraph(DT,true);    // display initial graph
#endif

#ifdef PRUNE
    edge e;
    forall_edges(e,DT)
    {
      if (DT[e] == HULL_EDGE)
```

```
        {
            char strBuffer[50];
                  ostrstream ostr(strBuffer,50);

                node v = DT.source(e), w = DT.target(e);
            point p1;
            p1 = DT[v].to_point();
            if (externalPoints.member(p1) == false)
                        externalPoints.insert(p1);
            ostr << p1;
            p1 = DT[w].to_point();
            if (externalPoints.member(p1) == false)
                        externalPoints.insert(p1);
            ostr << p1;
            ostr << '\0';
            externalEdges.insert(strBuffer);
        }
      }
#endif


      firstNode = currentNode = GetFirstNode(L,DT,startPoint);

#ifdef PRUNE
    point p1 = DT[firstNode].to_point();
      if (externalPoints.member(p1) == true)
      {
      edgeCounter = 1;
        connectList.append(1);
    }
      else
      {
      edgeCounter = 0;
        connectList.append(0);
    }
#endif


    remainingGraph = DT;        // copy graph into a remaining buffer
      previousGraph = DT;

    LPoint.append(DT[firstNode].to_point());    // store first point

#ifndef FAST_CODE
    if (graphicFlag)
      processMouse();
    else if (stepFlag)
    {
            cout << "Enter 's' to step\n" ;
            cin >> string1;
            cout << endl;
    }
#endif

    for (;;)
    {
            // get the list of adjacent nodes to the current node
        list<node> adjNodes = remainingGraph.adj_nodes(currentNode);
        point pStart,
              pEnd;
```

```
        int numAdjNodes = adjNodes.size();           // get the
number of adjacent nodes

        int flag = 0;
        int index = 0;
        node minNode;
        float minValue;
        float mainGraphMSTLength;
        point minPoint;

#ifndef FAST_CODE
        bool minUpdateFlag;
#endif

        float tempDistance = 0;
            GRAPH<point,int> tempGraphGraph;

        node tempNode;

        if (LPoint.size() > 1)
        {
                  int i,
                  j ;
            point x1,
                    x2;
            list_item listPtr;

                  j =  LPoint.size();

            x1 = LPoint.front();
            listPtr = LPoint.first();

            for ( i = 1 ; i < j ; i++)
            {
                  listPtr = LPoint.succ(listPtr);
                x2 = LPoint.contents(listPtr);
                tempDistance+= x1.distance(x2);
                x1 = x2;
            }
        }

        int Kounter = 0;
        forall(tempNode,adjNodes)
        {

            pStart = remainingGraph[currentNode].to_point();
            pEnd = remainingGraph[tempNode].to_point();

            if (isDeletedPoint(pEnd))
                  continue;

// check if endpoint has more that one adjacent node
              list<node> tempAdjNodes =
remainingGraph.adj_nodes(tempNode);

            node tempTempNode;

            int nodeCounter = 0;

#ifdef PRUNE
            bool externalEndPointFlag = externalPoints.member(pEnd);
```

```
            bool externalStartPointFlag =
externalPoints.member(pStart);

            int numAdjacentActiveHullPoints = 0;
#endif

            forall( tempTempNode,tempAdjNodes)
            {
                  point tPoint =
remainingGraph[tempTempNode].to_point();;

                if (isDeletedPoint(tPoint))
                        continue;

                        nodeCounter++;

#ifdef PRUNE
                if (externalEndPointFlag &&
externalPoints.member(tPoint))
                {
                  char strBuffer[50];
                              ostrstream ostr(strBuffer,40);
                        ostr << pEnd;
                        ostr << tPoint;
                      ostr << '\0';
                              if (externalEdges.member(strBuffer))
                           numAdjacentActiveHullPoints++;
                }
#endif
            }

                  if (LPoint.size()== numpoints -1)
                        nodeCounter++;


                  if (nodeCounter == 0)
                  continue;

#ifdef PRUNE
                  if (externalEndPointFlag && edgeCounter == 1 &&
numAdjacentActiveHullPoints == 2 && !externalStartPointFlag)
                        continue;
#endif

            Kounter++;

            float distance= pStart.distance(pEnd);

            float tempVal1 = distance +tempDistance +
pEnd.distance(LPoint.front()) -(tempDistance+
pStart.distance(LPoint.front()));

                  GRAPH<point,int> tempGraph;

                  tempGraph = remainingGraph;

            deleteNode(pStart,tempGraph);
                tempGraphGraph = tempGraph;


#ifndef FAST_CODE
```

```
            if (graphicFlag)
            {
                  if (flag == 0)
              {

                        list<point> newPointList;

                            node v;

                              forall_nodes(v,tempGraph)

      newPointList.append(tempGraph[v].to_point());

                        DELAUNAY_TRIANG(newPointList,tempGraph);

                      list<edge> el =
MIN_SPANNING_TREE(tempGraph,MSTcmp);

                    displayWholeGraph();
                    displayMinSpanningPath(tempGraph,el);

                        mainGraphMSTLength =
calculateMSTPathLength(tempGraph,el);

                        if (displayOutputFlag)
                                    cout<<"\nCalcuated MST for
remaining graph: " << mainGraphMSTLength<< endl;
                                if (outputFile)
                                    progOutput<<"\nCalcuated MST for
remaining graph: " << mainGraphMSTLength<< endl;

                        float  tempHeuristic =
getGraphUpdatedHeuristic(tempGraph,pStart);

                    if (tempHeuristic != 0)
                      {

                              if (displayOutputFlag)
                                          cout << "Updating Above
MST Heuristic Estimate From: "
                                      << mainGraphMSTLength
                                        << "  To: "
                                          << tempHeuristic
                                      << endl;

                                         if (outputFile)
                                               progOutput << "Updating
Above  MST Heuristic Estimate From: "
                                             << mainGraphMSTLength
                                             << "  To: "
                                          << tempHeuristic
                                            << endl;
                            remainingGraphHeuristicUpdates++;


                                    mainGraphMSTLength = tempHeuristic;
                    }
                        else

      remainingGraphNoHeuristicUpdates++;
```

```
                        processMouse();

                        tempGraph  = tempGraphGraph;
              }
                flag++;
            }
            else
#endif
            {
                  if (flag == 0)
              {

                    float  tempHeuristic =
getGraphUpdatedHeuristic(tempGraph,pStart);

                      if (tempHeuristic != 0)
                  {

#ifndef FAST_CODE
                       if (displayOutputFlag)
                                        cout << "Using Updated  MST
Heuristic Estimate: "
                                     << tempHeuristic
                                  << endl;

                                     if (outputFile)
                                          progOutput << "Using
Updated  MST Heuristic Estimate: "
                                          << tempHeuristic
                                      << endl;
                       remainingGraphHeuristicUpdates++;
#endif

                                mainGraphMSTLength = tempHeuristic;
                  }
                    else
                      {

#ifndef FAST_CODE

      remainingGraphNoHeuristicUpdates++;
#endif

                                list<point> newPointList;

                                    node v;

                                      forall_nodes(v,tempGraph)

      newPointList.append(tempGraph[v].to_point());


DELAUNAY_TRIANG(newPointList,tempGraph);

                            list<edge> el =
MIN_SPANNING_TREE(tempGraph,MSTcmp);


                         mainGraphMSTLength =
calculateMSTPathLength(tempGraph,el);
```

```
#ifndef FAST_CODE
                        if (displayOutputFlag)
                                        cout<<"\nCalcuated MST for
remaining graph: " << mainGraphMSTLength<< endl;
                                    if (outputFile)
                                        progOutput<<"\nCalcuated
MST for remaining graph: " << mainGraphMSTLength<< endl;
#endif


StoreGraphHeuristic(tempGraph,mainGraphMSTLength,pStart);
                                tempGraph  = tempGraphGraph;
                    }

#ifndef FAST_CODE
                               if (stepFlag)
                               {
                                      cout << "Enter 's' to step\n" ;
                                 cin >> string1;
                                      cout << endl;
                               }
#endif

                    }
                flag++;
            }


#ifndef FAST_CODE
            if (displayOutputFlag)
                        cout << "Start Point: "
                   << pStart
                     << "    Proposed Path End Point: "
                     << pEnd
                     << endl;
                   if (outputFile)
                        progOutput << "Start Point: "
                           << pStart
                             << "    Proposed Path End Point: "
                                 << pEnd
                             << endl;

            bool updateFlag = false;
#endif

            deleteNode(pEnd,tempGraph);

            float tempVal2;

#ifndef FAST_CODE
            if (graphicFlag)
            {
                  list<point> newPointList;

                      node v;

                        forall_nodes(v,tempGraph)
                        newPointList.append(tempGraph[v].to_point());

                  DELAUNAY_TRIANG(newPointList,tempGraph);
```

```
                displayWholeGraph();
              displayGraph(tempGraph,false);
                displayCurrentPath(LPoint);
                       displayProposedPath(pStart,pEnd);

                list<edge> el =
MIN_SPANNING_TREE(tempGraph,MSTcmp);

                displayMinSpanningPath(tempGraph,el);

                       tempVal2 =
calculateMSTPathLength(tempGraph,el);

                if (displayOutputFlag)
                           cout << "Loop distance is: "
                     << tempVal1
                       << "  MST of SubGraph is:  "
                   << tempVal2
                     << "  Total is: "
                       << (tempVal1 + tempVal2)
                   << endl;
                    if (outputFile)
                          progOutput << "Loop distance is: "
                               << tempVal1
                                 << "  MST of SubGraph is:  "
                             << tempVal2
                             << "  Total is: "
                                 << (tempVal1 + tempVal2)
                               << endl;


                float  updatedHeuristic =
getSubGraphUpdatedHeuristic(tempGraph,pEnd);
             if (updatedHeuristic)
               {

                       if (displayOutputFlag)
                                  cout << "Loop distance is: "
                            << tempVal1
                            << "  Heuristic Update of MST of
SubGraph is:  "
                        << updatedHeuristic
                          << "  Total is: "
                            << (tempVal1 + updatedHeuristic)
                            << endl;
                           if (outputFile)
                                 progOutput << "Loop distance is:
"
                                        << tempVal1
                                   << "  Heuristic Update of MST
of SubGraph is:  "
                                 << updatedHeuristic
                                   << "  Total is: "
                                     << (tempVal1 +
updatedHeuristic)
                                     << endl;
                 updateFlag = true;

                     tempVal2 = updatedHeuristic;
              }
                  }
```

```
                    else
#endif

                    {

                    float  updatedHeuristic =
getSubGraphUpdatedHeuristic(tempGraph,pEnd);
                if (updatedHeuristic != 0)
                  {

#ifndef FAST_CODE
                        if (displayOutputFlag)
                                    cout << "Loop distance is: "
                                << tempVal1
                                << "  Heuristic Update of MST of
SubGraph is:  "
                                << updatedHeuristic
                            << "  Total is: "
                              << (tempVal1 + updatedHeuristic)
                                << endl;
                               if (outputFile)
                                     progOutput << "Loop distance is:
"
                                       << tempVal1
                                     << "  Heuristic Update of MST of
SubGraph is:  "
                                       << updatedHeuristic
                                     << "  Total is: "
                                     << (tempVal1 + updatedHeuristic)
                                       << endl;
                        updateFlag = true;
#endif
                    tempVal2 = updatedHeuristic;
                  }
                    else
                    {
                          list<point> newPointList;

                              node v;

                                forall_nodes(v,tempGraph)

      newPointList.append(tempGraph[v].to_point());

                          DELAUNAY_TRIANG(newPointList,tempGraph);


                          list<edge> el =
MIN_SPANNING_TREE(tempGraph,MSTcmp);


                                    tempVal2 =
calculateMSTPathLength(tempGraph,el);

      StoreSubGraphHeuristic_2(tempGraphGraph,pEnd,tempVal2);

#ifndef FAST_CODE
                          if (displayOutputFlag)
                                      cout << "Loop distance is: "
                                       << tempVal1
                                  << "  MST of SubGraph is:  "
```

```
                                        << tempVal2
                                          << "  Total is: "
                                     << (tempVal1 + tempVal2)
                                          << endl;
                                    if (outputFile)
                                          progOutput << "Loop distance is:
"
                                             << tempVal1
                                             << "  MST of SubGraph is:  "
                                                    << tempVal2
                                                  << "  Total is: "
                                               << (tempVal1 + tempVal2)
                                                 << endl;
#endif
                              }
                        }


            float tempVal3 = tempVal1 + tempVal2;

            if (index == 0)
            {
                  minPoint = pEnd;
                        minNode = tempNode;
                  minValue =  tempVal3;
#ifndef FAST_CODE
                  minUpdateFlag = updateFlag;
#endif
            }
            else if (minValue  >  tempVal3)
            {
                  minPoint = pEnd;
                        minNode = tempNode;
                  minValue =  tempVal3;
#ifndef FAST_CODE
                  minUpdateFlag = updateFlag;
#endif
            }

            index++;

#ifndef FAST_CODE
                  if (graphicFlag)
                    processMouse();
                  else if (stepFlag)
                  {
                          cout << "Enter 's' to step\n" ;
                          cin >> string1;
                          cout << endl;
                  }
#endif
        }

        if (Kounter == 0)
        {
            point p1;

#ifdef PRUNE
            int connectListValue;
            connectListValue = connectList.back();
              connectList.Pop();
```

```
                edgeCounter -= connectListValue;
#endif

            tempGraphGraph = remainingGraph;

            pEnd = LPoint.back();
            deleteNode(pEnd,tempGraphGraph);
            StoreSubGraphHeuristic(tempGraphGraph,minPoint,MAXFLOAT);

            remainingGraph = graphStack.pop();
              LPoint.Pop();

                p1 = LPoint.back();

            currentNode= getGraphNodeFromPoint(remainingGraph,p1);

            pStart = LPoint.back();

#ifndef FAST_CODE
                if (graphicFlag)
            {
                  displayWholeGraph();
                    displayGraph(remainingGraph,false);    //
display initial graph
                  displayCurrentPath(LPoint);
            }

                if (graphicFlag)
                  processMouse();
                else if (stepFlag)
                {
                        cout << "Enter 's' to step\n" ;
                        cin >> string1;
                        cout << endl;
                }
#endif

            continue;
        }

#ifndef FAST_CODE
        if (minUpdateFlag)
            subGraphHeuristicUpdates++;
        else
                subGraphNoHeuristicUpdates++;


            if (displayOutputFlag)
        {
                 cout << "Main Graph MST: "
                  << mainGraphMSTLength
                << "  Min value is "
                << minValue
                << endl;

                 if (!altThresholdFlag && thresholdSelectedFlag)
                { cout << "Current Threshold: " << currentThreshold
<< endl;
              cout << "learning: "<< (minValue -
mainGraphMSTLength)<<endl;}
        }
```

```
            if (outputFile)
        {
                progOutput << "Main Graph MST: "
                        << mainGraphMSTLength
                        << "  Min value is "
                      << minValue
                   << endl;
                if (!altThresholdFlag && thresholdSelectedFlag)
                { cout << "Current Threshold: " << currentThreshold
<< endl;
               cout << "learning: "<< (minValue -
mainGraphMSTLength)<<endl;}
        }
#endif

            tempPreviousGraph = remainingGraph;
            graphStack.push(remainingGraph);

        deleteNode(pStart,remainingGraph);

        currentNode=minNode;
          LPoint.append(remainingGraph[minNode].to_point());    //
store first

#ifdef PRUNE
        int connectListValue = 0;
        if (externalPoints.member(minPoint) &&
!externalPoints.member(pStart))
             connectListValue = 1;

            edgeCounter += connectListValue;
        connectList.append(connectListValue);
#endif


// check for backtrack
            float thresholdCalc;
        if (altThresholdFlag)
                thresholdCalc =  mainGraphMSTLength *(1
+currentThreshold);
          else
            { if (!backtrackFlag)
                thresholdCalc =  mainGraphMSTLength
+currentThreshold;
                else thresholdCalc =  mainGraphMSTLength;}

            if (thresholdCalc <= minValue)
                point p1;

#ifndef FAST_CODE
                statsforwardFlag = false;
            if (!backtrackFlag)
                statsBacktrackSequenceCount++;
#endif
            backtrackFlag = true;

#ifdef PRUNE
            int connectListValue;
            connectListValue = connectList.back();
```

```
                connectList.Pop();
                    edgeCounter -= connectListValue;
#endif

              remainingGraph = graphStack.pop();
                LPoint.Pop();


                    p1 = LPoint.back();
              currentNode= getGraphNodeFromPoint(remainingGraph,p1);

              StoreSubGraphHeuristic(tempGraphGraph,minPoint,minValue);
              StoreGraphHeuristic(tempGraphGraph,minValue,pStart);

#ifndef FAST_CODE
              if ( backtrackFlag  && remainingGraph.number_of_nodes()
== numpoints)
                        statsBacktrackSequenceRootCount++;

              if (atRootFlag)
                        statsBacktrackRoot2Root++;
              atRootFlag = false;
                    statsBacktrackCount++;
#endif

                    if (remainingGraph.number_of_nodes() == numpoints)
              {

#ifndef FAST_CODE
                        if (displayOutputFlag)
                              cout << endl
                         << endl
                         << "-----Backtrack - Graphs the same to
point:"
                         << p1
                         << "-----"
                         << endl;
                        if (outputFile)
                              progOutput  << endl
                                    << endl
                                 << "-----Backtrack - Graphs the
same to point:"
                                   << p1
                                    << "-----"
                                    << endl;
                        if (graphicFlag)
                {
                          displayWholeGraph();
                            displayGraph(remainingGraph,false);    //
display initial graph
                        displayCurrentPath(LPoint);
                        processMouse();
                }
                      else if (stepFlag)
                      {
                              cout << "Enter 's' to step\n" ;
                              cin >> string1;
                              cout << endl;
                      }
                        atRootFlag = true;
#endif
```

```
                backtrackFlag = false;
              currentThreshold = initThreshold;
              continue;
          }


#ifdef PRUNE
          connectListValue = connectList.back();
            connectList.Pop();
              edgeCounter -= connectListValue;
#endif

          remainingGraph = graphStack.pop();
            LPoint.Pop();

              p1 = LPoint.back();
          currentNode= getGraphNodeFromPoint(remainingGraph,p1);

#ifndef FAST_CODE
              if (displayOutputFlag)
                    cout << endl
               << endl
                 << "-----Backtrack - POP twice to point: "
                 << p1
                 << "-----"
                 << endl;

              if (outputFile)
                    progOutput  << endl
                                << endl
                          << "-----Backtrack - POP twice to
point: "
                              << p1
                            << "-----"
                            << endl;

              if (graphicFlag)
          {
                  displayWholeGraph();
                    displayGraph(remainingGraph,false);    //
display initial graph
                  displayCurrentPath(LPoint);
                  processMouse();
          }
              else if (stepFlag)
              {
                      cout << "Enter 's' to step\n" ;
                      cin >> string1;
                      cout << endl;
              }
#endif
          continue;
      }

          previousGraph = tempPreviousGraph;

#ifndef FAST_CODE
      if (graphicFlag)
          processMouse();
        else if (stepFlag)
        {
```

```
                cout << "Enter 's' to step\n" ;
                cin >> string1;
                cout << endl;
      }
#endif

        if (altThresholdFlag)
                thresholdCalc =  mainGraphMSTLength *(1
+currentThreshold);
        else
            { if (!backtrackFlag)
                thresholdCalc =  mainGraphMSTLength
+currentThreshold;
                else thresholdCalc =  mainGraphMSTLength;}

            if (thresholdCalc >= minValue && LPoint.size() ==
numpoints)
        {

                time_t endTime = time(NULL);

            float pathDistance = redLineLength();
            int i,j;
            i = endTime;
            j =  startTime;
            cout << "Number of Points: " << numpoints  << endl;
            cout << "Real Run Time: " << (i - j)  << endl;
            cout << "LEDA time calculation: " << used_time()<< endl;


#ifndef FAST_CODE
                if (outputFile)
            {
                  progOutput << "Number of Points: " << numpoints
<< endl;
                      progOutput  << "Real Run Time: " << (i - j)
<< endl;
                progOutput << " LEDA time calculation: " <<
used_time()<< endl;
            }
#endif

#ifndef __BORLANDC__
            struct tms timeBuff;
                times(&timeBuff);

            double xx = timeBuff.tms_utime;

            double userTime = xx/CLOCKS_PER_SEC;

            cout << "CPU User Time: " << userTime  << endl;

#ifndef FAST_CODE
                if (outputFile)
                    progOutput << "CPU User Time: " << userTime
<< endl;
#endif
            xx = timeBuff.tms_stime;

            double sysTime = xx/CLOCKS_PER_SEC;
```

```
            cout << "OS System Time: " << sysTime  << endl;

#ifndef FAST_CODE
                  if (outputFile)
                        progOutput <<  "OS System Time: " << sysTime
<< endl;
#endif
#endif

                  cout << "Minimum Path ";
#ifndef FAST_CODE
                  if (outputFile)
                        progOutput  << "Minimum Path ";
#endif
                  point xxxx;
                  forall(xxxx,LPoint)
            {
                        cout << xxxx;
#ifndef FAST_CODE
                        if (outputFile)
                              progOutput << xxxx;
#endif
            }
                  cout << LPoint.front() << endl;

#ifndef FAST_CODE
                  if (outputFile)
                        progOutput << LPoint.front() << endl;
#endif

               LPoint.append( LPoint.front());
            cout << "Path length is: " << pathDistance << "   Closed
Loop distance is: " << redLineLength() <<endl;
            cout << "Init Threshold Value: "<< initThreshold << endl;

#ifndef FAST_CODE
            if (statisticsFlag)
            {
                        cout << "Forward operations: " <<
statsFowardCount << "   Forward Sequences: " <<
statsForwardSequenceCount << endl;
                        cout << "Backtrack operations: " <<
statsBacktrackCount << "   Backtrack Sequences: " <<
statsBacktrackSequenceCount << endl;

                        cout << "Backtrack to root: " <<
statsBacktrackSequenceRootCount << "   Backtrack root to root: " <<
statsBacktrackRoot2Root << endl;

                  cout << "Heuristic Updates Used in Remaining Graph:
" << remainingGraphHeuristicUpdates << endl;
               cout <<  "No Heuristic Updates Used in Remaining Graph:
" << remainingGraphNoHeuristicUpdates << endl;

                cout << "Heuristic Updates Used in Sub-Graph: " <<
subGraphHeuristicUpdates << endl;
                  cout <<  "No Heuristic Updates Used in Sub-Graph: "
<< subGraphNoHeuristicUpdates << endl;

#ifdef NODE_TABLES
                cout << "Remaining Graph Bin Statistics" << endl;
```

```
                int i;

                for ( i = 0 ; i <= numpoints ; i++)
                {
                  int sum = 0;
                              struct binNode *binNodePtr;

                              binNodePtr = (graphLoopUpTablePtr + i)-
>binNodePtr;

                    while ( binNodePtr)
                    {
                        sum += binNodePtr->count;
                                    binNodePtr = binNodePtr->nextBin;
                    }

                    cout << i << "-" << sum << "   ";
                }
                cout << endl;

                cout << "SubGraph Bin Statistics" << endl;

                for ( i = 0 ; i <= numpoints ; i++)
                {
                  int sum = 0;
                              struct binNode *binNodePtr;

                              binNodePtr = (subGraphLoopUpTablePtr +
i)->binNodePtr;

                    while ( binNodePtr)
                    {
                        sum += binNodePtr->count;
                                    binNodePtr = binNodePtr->nextBin;
                    }

                    cout << i << "-" << sum << "   ";
                }
                cout << endl;
#endif
            }
#endif

            cout << "Program Terminated\n" << endl;

#ifndef FAST_CODE
                  if (outputFile)
                  {
              progOutput << "Path length is: " << pathDistance << "
Closed Loop distance is: " << redLineLength() <<endl;
                  progOutput << "Init Threshold Value: "<<
initThreshold << endl;
                  if (statisticsFlag)
              {
                              progOutput << "Forward operations: " <<
statsFowardCount << "   Forward Sequences: " <<
statsForwardSequenceCount << endl;
                              progOutput << "Backtrack operations: "
<< statsBacktrackCount << "   Backtrack Sequences: " <<
statsBacktrackSequenceCount << endl;
```

```
                                progOutput << "Backtrack to root: " <<
statsBacktrackSequenceRootCount << "   Backtrack root to root: " <<
statsBacktrackRoot2Root << endl;

                          progOutput << "Heuristic Updates Used in
Remaining Graph: " << remainingGraphHeuristicUpdates << endl;
                         progOutput <<  "No Heuristic Updates Used in
Remaining Graph: " << remainingGraphNoHeuristicUpdates << endl;

                        progOutput << "Heuristic Updates Used in Sub-
Graph: " << subGraphHeuristicUpdates << endl;
                          progOutput <<  "No Heuristic Updates Used in
Sub-Graph: " << subGraphNoHeuristicUpdates << endl;

#ifdef NODE_TABLES
                        progOutput << "Remaining Graph Bin Statistics"
<< endl;
                    int i;

                      for ( i = 0 ; i <= numpoints ; i++)
                        {
                          int sum = 0;
                                    struct binNode *binNodePtr;

                                       binNodePtr = (graphLoopUpTablePtr +
i)->binNodePtr;

                        while ( binNodePtr)
                          {
                                sum += binNodePtr->count;
                                           binNodePtr = binNodePtr-
>nextBin;
                            }

                          progOutput << i << "-" << sum << "   ";
                        }
                        progOutput << endl;

                    progOutput << "SubGraph Bin Statistics" << endl;

                      for ( i = 0 ; i <= numpoints ; i++)
                        {
                          int sum = 0;
                                    struct binNode *binNodePtr;

                                       binNodePtr =
(subGraphLoopUpTablePtr + i)->binNodePtr;

                        while ( binNodePtr)
                          {
                                sum += binNodePtr->count;
                                           binNodePtr = binNodePtr-
>nextBin;
                            }

                        progOutput << i << "-" << sum << "   ";
                       }
                        progOutput << endl;
#endif
                    }
```

```
                progOutput << "Program Terminated" << endl;
            }

                if (endOfRunGraph )
            {
                    graphicFlag = true;
                  stepBut = WDD.button("STEP");
                clearBut = WDD.button("CLEAR");
                    autoBut = WDD.button("AUTO");
                exitBut = WDD.button("EXIT");

                    FindScalingFactors(&xfact,&yfact);
                WDD.init(minX-xfact,maxX+xfact,minY-yfact);
                WDD.display(window::center,window::center);
                WDD.start_buffering();
            }

          if (graphicFlag)
          {
                    autoFlag = false;
                  displayWholeGraph();
                displayGraph(remainingGraph,false);    // display
initial graph
                displayCurrentPath(LPoint);
              processMouse();
                    WDD.stop_buffering();
            }
#endif

            return 0;
        }


        if (!altThresholdFlag )
        {
            if (!backtrackFlag)
            {
                if ( mainGraphMSTLength < minValue)
                    currentThreshold = currentThreshold -
minValue+ mainGraphMSTLength;
            }
            else
              { if (    mainGraphMSTLength > minValue)
                    currentThreshold = initThreshold;
    //              if  ( mainGraphMSTLength <= minValue)
    //                    currentThreshold = 0.0;
            }
        }

#ifndef FAST_CODE
        if (graphicFlag)
        {
            displayWholeGraph();
              displayGraph(remainingGraph,false);    // display
initial graph
            displayCurrentPath(LPoint);
            processMouse();
        }
          else if (stepFlag)
      {
                cout << "Enter 's' to step\n" ;
```

```
                cin >> string1;
                cout << endl;
        }

          if (displayOutputFlag)
                cout << endl
                 << endl
                << "------ Move Forward -------"
                << endl;
          if (outputFile)
                progOutput << endl
                           << endl
                           << "------ Move Forward -------"
                          << endl;

          if (!statsforwardFlag)
                statsForwardSequenceCount++;
        statsforwardFlag = true;
        atRootFlag = false;
        statsFowardCount++;
#endif
          backtrackFlag = false;

    }
    return 0;
}


#ifndef FAST_CODE
void processMouse(void)
{
    int but;
    if ( autoFlag && WDD.get_mouse() != NO_BUTTON)
    {
          but = WDD.read_mouse();
        if (but == exitBut)
          exit(0);
        else if (but == autoBut)
        {
                autoFlag = !autoFlag;
          return;
        }
        else if (but == clearBut)
        {
          WDD.clear();
          return;
        }
        else
        {
                WDD.flush_buffer();
          return;
        }
      }

      if ( autoFlag)
    {
          WDD.flush_buffer();
          time_t ti;
        ti = time(NULL);
        ti +=delayGap;
        while (ti > time(NULL))
```

```
        ;
        return;
    }

      while (true)
    {
            but = WDD.read_mouse();

        if (but == exitBut)
            exit(0);
        else if (but == autoBut)
        {
                  autoFlag = !autoFlag;
            return;
        }
        else if (but == clearBut)
        {
            WDD.clear();
            return;
        }
        else
        {
                  WDD.flush_buffer();
            return;
        }
      }
}

void FindScalingFactors(float *xfact,float *yfact)
{
// Calculates a value for a scaling factor that will allow
// the whole graph to be displayed in a window

    *xfact = (maxX -minX)/100*5;
    *yfact = (maxY - minY)/100*10;
    float xgap = maxX-minX+2* *xfact;
    float ygap = maxY-minY+2* *yfact;

    float sfact = WDD.height()/ygap;

    float cal_xgap = WDD.width()/sfact;

    if (xgap < cal_xgap)
            *xfact += (cal_xgap - xgap)/2;
      return;
}

#endif

node GetFirstNode(list<point>& L,GRAPH<point,int>& DT, int
startPoint)
{
      static node v;
    point p;

      int i = 1;
    forall(p,L)
    {
      if ( i++ == startPoint)
            break;
    }
```

```
    forall_nodes(v,DT)
        if ( DT[v].to_point() == p)
            return v;

      return NULL;
}


void deleteNode(point& p,GRAPH<point,int>& G)
{
    node v;
    list<edge> el;

    forall_nodes(v,G)
    {
        if (G[v].to_point() ==  p)
        {
// removed to giveabout .5% gain
//                el = G.adj_edges(v);
//          G.del_edges(el);
            G.del_node(v);
            break;
        }
    }
      return;
}

#ifndef FAST_CODE
void displayGraph(GRAPH<point,int>& G,bool clearFlag)
{
    node v;
    edge e;

      WDD.set_line_width(1);
    WDD.set_node_width(4);

    WDD.set_color(COLOR_NORMAL);

    if (clearFlag)
            WDD.clear();

      forall_nodes(v,G)
            WDD.draw_filled_node(G[v].to_point());

      forall_edges(e,G)
      {
         node v = G.source(e), w = G.target(e);
         WDD.draw_segment(G[v].to_point(),G[w].to_point());
      }

      WDD.flush_buffer();
}


void displayWholeGraph(void )
{
    node v;
    edge e;

      WDD.set_line_width(1);
```

```
    WDD.set_node_width(4);

    WDD.set_color(COLOR_DELETED_PATH);

      WDD.clear();

      forall_nodes(v,DT)
            WDD.draw_filled_node(DT[v].to_point());

      forall_edges(e,DT)
      {
         node v = DT.source(e), w = DT.target(e);
         WDD.draw_segment(DT[v].to_point(),DT[w].to_point());
      }

      WDD.flush_buffer();
}


void  displayProposedPath(point& p1,point& p2)
{

      WDD.set_line_width(4);
    WDD.set_color(COLOR_PROPOSED_PATH);
    WDD.draw_segment(p1,p2);
      WDD.flush_buffer();
}


void  displayCurrentPath(list<point>& listPoints)
{
      int i= 0;

    if (listPoints.length() <= 1)
      return;

      WDD.set_line_width(4);
    WDD.set_color(COLOR_CURRENT_PATH);

    point p1,p2,x;
    forall(x,listPoints)
    {
            if (i == 0)
        {
                  p1 = x;
              i++;
            continue;
        }
        i++;
        p2 = x;
          WDD.draw_segment(p1,p2);
        p1 = p2;
    }
      WDD.flush_buffer();
}


void displayMinSpanningPath(GRAPH<point,int>& G, list<edge>& el)
{
      edge e;
      WDD.set_line_width(4);
```

```
    WDD.set_color(COLOR_MIN_PATH);

      forall(e,el)
      {
         node v = G.source(e), w = G.target(e);
         WDD.draw_segment(G[v].to_point(),G[w].to_point());
    }
      WDD.flush_buffer();
}

#endif

float calculateMSTPathLength(GRAPH<point,int>& G, list<edge>& el)
{
    point p1s,p1t;
    node n1s,n1t;
    float dist = 0.0;
    edge e;

      forall(e,el)
      {
          n1s = source(e);
      n1t = target(e);
          p1s = G[n1s].to_point();
          p1t = G[n1t].to_point();
            dist += p1s.distance(p1t);
    }
    return dist;
}


int MSTcmp(const edge &e1,const edge &e2)
{
      GRAPH<point,int> *G;
    node n1s,n1t,n2s,n2t;
    point p1s,p1t,p2s,p2t;

    G = (GRAPH<point,int> *)graph_of(e1);

    n1s = source(e1);
    n2s = source(e2);

    n1t = target(e1);
    n2t = target(e2);

    p1s = (*G)[n1s].to_point();
    p1t = (*G)[n1t].to_point();
    p2s = (*G)[n2s].to_point();
    p2t = (*G)[n2t].to_point();

      return cmp_distances(p1s,p1t,p2s,p2t);
}


int isDeletedPoint(point &pEnd)
{
      point x;
    forall(x,LPoint)
    {
            if (x == pEnd)
                  return 1;
```

```
    }
      return 0;
}


node getGraphNodeFromPoint(GRAPH<point,int>& G,point &p1)
{
    node v;

      forall_nodes(v,G)
    {
          if (G[v].to_point() == p1)
            return v;
    }
      cerr << "Point not in graph" << endl;
    exit(1);
      return v;
}


void InitLookUpTables(void)
{
      graphLoopUpTablePtr = new  heuristicLookUpTable[numpoints+1];
    if (!graphLoopUpTablePtr)
      {
            cerr << endl << "NO MEMORY FOR LOOKUP TABLE (1)" << endl
;
        exit(1);
    }
      memset((void
*)graphLoopUpTablePtr,'\0',sizeof(heuristicLookUpTable) *
(numpoints+1));


#ifdef NODE_TABLES
      tempGraphHeuristicNodePtr = new heuristicNode[bin_size];

    if (!tempGraphHeuristicNodePtr)
      {
            cerr << endl << "NO MEMORY FOR TEMP HEURISTIC (2)" <<
endl ;
        exit(1);
      }

#endif

      subGraphLoopUpTablePtr = new
heuristicLookUpTable[numpoints+1];
    if (!subGraphLoopUpTablePtr)
      {
            cerr << endl << "NO MEMORY FOR LOOKUP TABLE (3)" << endl
;
        exit(1);
    }
      memset((void
*)subGraphLoopUpTablePtr,'\0',sizeof(heuristicLookUpTable) *
(numpoints+1));


#ifdef NODE_TABLES
      tempSubGraphHeuristicNodePtr = new heuristicNode[bin_size];
```

```
    if (!tempSubGraphHeuristicNodePtr)
      {
            cerr << endl << "NO MEMORY FOR TEMP HEURISTIC (4)" <<
endl ;
        exit(1);
      }
#endif
}



#ifdef NODE_TABLES
void StoreGraphHeuristic(GRAPH<point,int>& G,float heuristic,point&
startPoint)
{
      int numOfNodes = G.number_of_nodes();


      float *ptsBuffer = new float[LPoint.size() * 2];
    if (!ptsBuffer)
      {
            cerr << endl << "NO MEMORY FOR POINT BUFFER (5)" << endl
;
        exit(1);
    }


      point x;

    int i = 0;
    forall(x,LPoint)
    {
        *(ptsBuffer + i) = x.xcoord();
        i++;
        *(ptsBuffer + i) = x.ycoord();
        i++;
    }

      struct binNode *binNodePtr;

    binNodePtr = (graphLoopUpTablePtr + numOfNodes)->binNodePtr;

    if (binNodePtr == NULL)
    {
      binNodePtr = (graphLoopUpTablePtr + numOfNodes)->binNodePtr =
new binNode;
          if (!binNodePtr)
            {
                  cerr << endl << "NO MEMORY FOR BIN NODE (6)" <<
endl ;
              exit(1);
      }
            binNodePtr->count = 1;
        binNodePtr->nextBin = NULL;

        binNodePtr->heuristicNodePtr = new heuristicNode[bin_size];

          if (!binNodePtr->heuristicNodePtr)
            {
```

```
                cerr << endl << "NO MEMORY FOR BIN NODE (7)" <<
endl ;
            exit(1);
      }
        memset(binNodePtr-
>heuristicNodePtr,'\0',sizeof(heuristicNode)*bin_size);
        binNodePtr->heuristicNodePtr[0].heuristic = heuristic;
        binNodePtr->heuristicNodePtr[0].graphPtr = ptsBuffer;
    }
    else
          StoreGraphUpdatedHeuristic(binNodePtr,heuristic,ptsBuffer);
}


void StoreGraphUpdatedHeuristic(struct binNode *binNodePtr,float
heuristic,float *ptsBuffer)
{
      int pos;

      while (binNodePtr)
      {
            if ((pos = myGraphBsearch(ptsBuffer,
binNodePtr,LPoint.size())) == -1)
            {
                  if (binNodePtr->count == bin_size)
                  {
                        if (binNodePtr->nextBin == NULL)
                        {
                              if ((binNodePtr->nextBin = new binNode)
== NULL)
                              {
                                    cerr << endl << "NO MEMORY FOR
BIN NODE (8)" << endl ;
                                    exit(1);
                              }

                              binNodePtr->nextBin->count = 1;
                          binNodePtr->nextBin->nextBin = NULL;

                          binNodePtr->nextBin->heuristicNodePtr = new
heuristicNode[bin_size];

                            if (!binNodePtr->nextBin-
>heuristicNodePtr)
                              {
                                    cerr << endl << "NO MEMORY FOR
BIN NODE (9)" << endl ;
                                exit(1);
                              }
                          memset(binNodePtr->nextBin-
>heuristicNodePtr,'\0',sizeof(heuristicNode)*bin_size);
                         binNodePtr->nextBin-
>heuristicNodePtr[0].heuristic = heuristic;
                        binNodePtr->nextBin-
>heuristicNodePtr[0].graphPtr = ptsBuffer;
                      return;
                        }

                        binNodePtr = binNodePtr->nextBin;
                        continue;
                  }
```

```
      InsertGraphUpdatedHeuristic(binNodePtr,heuristic,ptsBuffer);
            return;
            }
            else
        {
                if (binNodePtr->heuristicNodePtr[pos].heuristic <
heuristic)
                      binNodePtr->heuristicNodePtr[pos].heuristic =
heuristic;
            return;
        }
      }
      cerr << endl << "IMPOSSIBLE CONDITION (10)" << endl ;
    exit(1);
}


void InsertGraphUpdatedHeuristic(struct binNode *binNodePtr,float
heuristic, float *ptsBuffer)
{
      int low,
            high,
            mid,
            result;

// set up a binary search to find the insertion point
      high = binNodePtr->count -1;
      low = 0;

      while (low <= high)
      {
            mid = (high + low)/2;
            result = cmpPts(ptsBuffer,binNodePtr-
>heuristicNodePtr[mid].graphPtr,LPoint.size());
            if (result < 0)
                  high = mid -1;
            else if (result > 0)
                  low = mid + 1;
            else
            {
// the condition is impossible and indicates a major problem
                  cerr << endl << "IMPOSSIBLE CONDITION  (11)" <<
endl ;
                  exit(1);
            }
       }

// move elements from insertion point to a holding buffer
      memmove(tempGraphHeuristicNodePtr,&binNodePtr-
>heuristicNodePtr[low],(binNodePtr->count - low)*sizeof(struct
heuristicNode));

      binNodePtr->heuristicNodePtr[low].heuristic = heuristic;
    binNodePtr->heuristicNodePtr[low].graphPtr = ptsBuffer;

// move schedule nodes back to the backtrack position plus one
      memmove(&binNodePtr-
>heuristicNodePtr[low+1],tempGraphHeuristicNodePtr,(binNodePtr->count
- low)*sizeof(struct heuristicNode));
```

```
// increament the count of schedule nodes in the current backtrack
node
      binNodePtr->count++;
      return;
}


int myGraphBsearch(float *ptsBuffer, struct binNode *binNodePtr,int
size)
{
// binary search the backtrack node for the key
// return -1 if not found or keys position in backtrack node
      int low,
            high,
            mid,
            result;

      high = binNodePtr->count -1;
      low = 0;

      while (low <= high)
      {
            mid = (high + low)/2;
            result = cmpPts(ptsBuffer,binNodePtr-
>heuristicNodePtr[mid].graphPtr,size);
            if (result < 0)
                  high = mid -1;
            else if (result > 0)
                  low = mid + 1;
            else
                  return mid;
      }
      return(-1);
}



float getGraphUpdatedHeuristic(GRAPH<point,int>& G, point&
startPoint)
{
      int numOfNodes = G.number_of_nodes();

      struct binNode *binNodePtr;

    binNodePtr = (graphLoopUpTablePtr + numOfNodes)->binNodePtr;
    if (binNodePtr == NULL)
            return 0;

      float *ptsBuffer = new float[LPoint.size() * 2];
    if (!ptsBuffer)
      {
            cerr << endl << "NO MEMORY FOR POINT BUFFER (12)" << endl
;
        exit(1);
    }


      point x;

    int i = 0;
    forall(x,LPoint)
```

```
    {
        *(ptsBuffer + i) = x.xcoord();
        i++;
        *(ptsBuffer + i) = x.ycoord();
        i++;
    }

      int pos;

      while (binNodePtr)
      {
            if ((pos = myGraphBsearch(ptsBuffer,
binNodePtr,LPoint.size())) != -1)
        {
            delete ptsBuffer;
                return binNodePtr->heuristicNodePtr[pos].heuristic;
        }
            binNodePtr = binNodePtr->nextBin;
      }
    delete ptsBuffer;
      return 0;
}

//*****************************************************************
*************

void StoreSubGraphHeuristic(GRAPH<point,int>& G,point& endPoint,float
heuristic)
{

      int numOfNodes = G.number_of_nodes();


      float *ptsBuffer = new float[LPoint.size() * 2];
    if (!ptsBuffer)
      {
            cerr << endl << "NO MEMORY FOR POINT BUFFER (13)" << endl
;
        exit(1);
    }


      point x;

    int i = 0;
    forall(x,LPoint)
    {
        *(ptsBuffer + i) = x.xcoord();
        i++;
        *(ptsBuffer + i) = x.ycoord();
        i++;
    }


      struct binNode *binNodePtr;

    binNodePtr = (subGraphLoopUpTablePtr + numOfNodes)->binNodePtr;

    if (binNodePtr == NULL)
    {
```

```
      binNodePtr = (subGraphLoopUpTablePtr + numOfNodes)->binNodePtr
= new binNode;
          if (!binNodePtr)
            {
                  cerr << endl << "NO MEMORY FOR BIN NODE (14)" <<
endl ;
              exit(1);
      }
            binNodePtr->count = 1;
        binNodePtr->nextBin = NULL;

        binNodePtr->heuristicNodePtr = new heuristicNode[bin_size];

          if (!binNodePtr->heuristicNodePtr)
            {
                  cerr << endl << "NO MEMORY FOR BIN NODE (15)" <<
endl ;
              exit(1);
      }
        memset(binNodePtr-
>heuristicNodePtr,'\0',sizeof(heuristicNode)*bin_size);

        binNodePtr->heuristicNodePtr[0].heuristic = heuristic;
        binNodePtr->heuristicNodePtr[0].graphPtr = ptsBuffer;
    }
    else

StoreSubGraphUpdatedHeuristic(binNodePtr,heuristic,ptsBuffer);
}


void StoreSubGraphHeuristic_2(GRAPH<point,int>& G,point&
endPoint,float heuristic)
{

      int numOfNodes = G.number_of_nodes();


      float *ptsBuffer = new float[LPoint.size() * 2 + 2];
    if (!ptsBuffer)
      {
            cerr << endl << "NO MEMORY FOR POINT BUFFER (16)" << endl
;
        exit(1);
    }


      point x;

    int i = 0;
    forall(x,LPoint)
    {
        *(ptsBuffer + i) = x.xcoord();
        i++;
        *(ptsBuffer + i) = x.ycoord();
        i++;
    }

    *(ptsBuffer + i) = endPoint.xcoord();
    i++;
    *(ptsBuffer + i) = endPoint.ycoord();
```

```
    i++;

      struct binNode *binNodePtr;

    binNodePtr = (subGraphLoopUpTablePtr + numOfNodes)->binNodePtr;

    if (binNodePtr == NULL)
    {
      binNodePtr = (subGraphLoopUpTablePtr + numOfNodes)->binNodePtr
= new binNode;
          if (!binNodePtr)
            {
                  cerr << endl << "NO MEMORY FOR BIN NODE (17)" <<
endl ;
              exit(1);
      }
            binNodePtr->count = 1;
        binNodePtr->nextBin = NULL;

        binNodePtr->heuristicNodePtr = new heuristicNode[bin_size];

          if (!binNodePtr->heuristicNodePtr)
            {
                  cerr << endl << "NO MEMORY FOR BIN NODE (18)" <<
endl ;
              exit(1);
      }
        memset(binNodePtr-
>heuristicNodePtr,'\0',sizeof(heuristicNode)*bin_size);

        binNodePtr->heuristicNodePtr[0].heuristic = heuristic;
        binNodePtr->heuristicNodePtr[0].graphPtr = ptsBuffer;
    }
    else

StoreSubGraphUpdatedHeuristic(binNodePtr,heuristic,ptsBuffer);
}



void StoreSubGraphUpdatedHeuristic(struct binNode *binNodePtr,float
heuristic,float *ptsBuffer)
{
      int pos;

      while (binNodePtr)
      {
            if ((pos =
mySubGraphBsearch(ptsBuffer,binNodePtr,LPoint.size())) == -1)
            {
                  if (binNodePtr->count == bin_size)
                  {
                        if (binNodePtr->nextBin == NULL)
                        {
                              if ((binNodePtr->nextBin = new binNode)
== NULL)
                              {
                                    cerr << endl << "NO MEMORY FOR
BIN NODE (19)" << endl ;
                                    exit(1);
```

```
                                }

                                    binNodePtr->nextBin->count = 1;
                                binNodePtr->nextBin->nextBin = NULL;

                                binNodePtr->nextBin->heuristicNodePtr = new
heuristicNode[bin_size];

                                  if (!binNodePtr->nextBin-
>heuristicNodePtr)
                                    {
                                          cerr << endl << "NO MEMORY FOR
BIN NODE (20)" << endl ;
                                      exit(1);
                                    }
                                 memset(binNodePtr->nextBin-
>heuristicNodePtr,'\0',sizeof(heuristicNode)*bin_size);
                                binNodePtr->nextBin-
>heuristicNodePtr[0].heuristic = heuristic;
                              binNodePtr->nextBin-
>heuristicNodePtr[0].graphPtr = ptsBuffer;
                            return;
                                }

                                binNodePtr = binNodePtr->nextBin;
                                continue;
                        }

      InsertSubGraphUpdatedHeuristic(binNodePtr,heuristic,ptsBuffer);
            return;
            }
            else
        {
                if (binNodePtr->heuristicNodePtr[pos].heuristic <
heuristic)
                      binNodePtr->heuristicNodePtr[pos].heuristic =
heuristic;
            return;
        }
      }
      cerr << endl << "IMPOSSIBLE CONDITION (21)" << endl ;
    exit(1);
}


void InsertSubGraphUpdatedHeuristic(struct binNode *binNodePtr,float
heuristic, float *ptsBuffer)
{
      int low,
            high,
            mid,
            result;

// set up a binary search to find the insertion point
      high = binNodePtr->count -1;
      low = 0;

      while (low <= high)
      {
            mid = (high + low)/2;
```

```
            result = cmpPts(ptsBuffer,binNodePtr-
>heuristicNodePtr[mid].graphPtr,LPoint.size());
            if (result < 0)
                  high = mid -1;
            else if (result > 0)
                  low = mid + 1;
            else
            {
// the condition is impossible and indicates a major problem
                  cerr << endl << "IMPOSSIBLE CONDITION (22)" << endl
;
                  exit(1);
            }
      }

// move elements from insertion point to a holding buffer
      memmove(tempSubGraphHeuristicNodePtr,&binNodePtr-
>heuristicNodePtr[low],(binNodePtr->count - low)*sizeof(struct
heuristicNode));

      binNodePtr->heuristicNodePtr[low].heuristic = heuristic;
    binNodePtr->heuristicNodePtr[low].graphPtr = ptsBuffer;

// move schedule nodes back to the backtrack position plus one
      memmove(&binNodePtr-
>heuristicNodePtr[low+1],tempSubGraphHeuristicNodePtr,(binNodePtr-
>count - low)*sizeof(struct heuristicNode));

// increament the count of schedule nodes in the current backtrack
node
      binNodePtr->count++;
      return;
}


int mySubGraphBsearch(float *ptsBuffer, struct binNode
*binNodePtr,int size)
{
// binary search the backtrack node for the key
// return -1 if not found or keys position in backtrack node
      int low,
            high,
            mid,
            result;

      high = binNodePtr->count -1;
      low = 0;

      while (low <= high)
      {
            mid = (high + low)/2;
            result = cmpPts(ptsBuffer,binNodePtr-
>heuristicNodePtr[mid].graphPtr,size);
            if (result < 0)
                  high = mid -1;
            else if (result > 0)
                  low = mid + 1;
            else
                  return mid;
      }
      return(-1);
```

```
}



float getSubGraphUpdatedHeuristic(GRAPH<point,int>& G, point&
endPoint)
{

      int numOfNodes = G.number_of_nodes();
      struct binNode *binNodePtr;

    binNodePtr = (subGraphLoopUpTablePtr + numOfNodes)->binNodePtr;

    if (binNodePtr == NULL)
            return 0;

      float *ptsBuffer = new float[LPoint.size() * 2 + 2];
    if (!ptsBuffer)
      {
            cerr << endl << "NO MEMORY FOR POINT BUFFER (23)" << endl
;
        exit(1);
    }


      point x;

    int i = 0;
    forall(x,LPoint)
    {
        *(ptsBuffer + i) = x.xcoord();
        i++;
        *(ptsBuffer + i) = x.ycoord();
        i++;
    }
    *(ptsBuffer + i) = endPoint.xcoord();
    i++;
    *(ptsBuffer + i) = endPoint.ycoord();
    i++;


      int pos;

      while (binNodePtr)
      {
            if ((pos = mySubGraphBsearch(ptsBuffer,
binNodePtr,LPoint.size()+1)) != -1)
        {
            delete ptsBuffer;
                return binNodePtr->heuristicNodePtr[pos].heuristic;
        }
            binNodePtr = binNodePtr->nextBin;
      }
    delete ptsBuffer;
      return 0;
}

#else
void StoreGraphHeuristic(GRAPH<point,int>& G,float heuristic,point
&p)
{
```

```
      int numOfNodes = G.number_of_nodes();

      float *ptsBuffer = new float[LPoint.size() * 2];
    if (!ptsBuffer)
      {
            cerr << endl << "NO MEMORY FOR POINT BUFFER (24)" << endl
;
        exit(1);
    }


      point x;

    int i = 0;
    forall(x,LPoint)
    {
        *(ptsBuffer + i) = x.xcoord();
        i++;
        *(ptsBuffer + i) = x.ycoord();
        i++;
    }

      struct binNode *binNodePtr;

    binNodePtr = (graphLoopUpTablePtr + numOfNodes)->binNodePtr;

    if (binNodePtr == NULL)
    {
      binNodePtr = (graphLoopUpTablePtr + numOfNodes)->binNodePtr =
new binNode;
          if (!binNodePtr)
            {
                  cerr << endl << "NO MEMORY FOR BIN NODE (25)" <<
endl ;
              exit(1);
       }
            binNodePtr->left = NULL;
        binNodePtr->right = NULL;
            binNodePtr->heuristic = heuristic;
            binNodePtr->graphPtr = ptsBuffer;
    }
    else
          StoreGraphUpdatedHeuristic(binNodePtr,heuristic,ptsBuffer);
}


void StoreGraphUpdatedHeuristic(struct binNode *binNodePtr,float
heuristic,float *ptsBuffer)
{
    struct binNode *tempPtr;
    int result;

      while (binNodePtr)
      {
            result = cmpPts(ptsBuffer,binNodePtr-
>graphPtr,LPoint.size());
        tempPtr = binNodePtr;
        if (result == 0)
        {
                if (binNodePtr->heuristic < heuristic)
                      binNodePtr->heuristic = heuristic;
```

```
                return;
        }
        else if (result == -1)
                binNodePtr = binNodePtr->left;
        else
                binNodePtr = binNodePtr->right;
      }

      binNodePtr = new binNode;
    if (!binNodePtr)
      {
            cerr << endl << "NO MEMORY FOR BIN NODE (26)" << endl ;
        exit(1);
      }
      binNodePtr->left = NULL;
    binNodePtr->right = NULL;
      binNodePtr->heuristic = heuristic;
      binNodePtr->graphPtr = ptsBuffer;
    if (result == -1)
      tempPtr->left = binNodePtr;
    else
      tempPtr->right = binNodePtr;
      return;
}


float getGraphUpdatedHeuristic(GRAPH<point,int>& G, point&
startPoint)
{
      int numOfNodes = G.number_of_nodes();

      struct binNode *binNodePtr;

    binNodePtr = (graphLoopUpTablePtr + numOfNodes)->binNodePtr;
    if (binNodePtr == NULL)
            return 0;

      float *ptsBuffer = new float[LPoint.size() * 2];
    if (!ptsBuffer)
      {
            cerr << endl << "NO MEMORY FOR POINT BUFFER (27)" << endl
;
        exit(1);
    }


      point x;

    int i = 0;
    forall(x,LPoint)
    {
        *(ptsBuffer + i) = x.xcoord();
        i++;
        *(ptsBuffer + i) = x.ycoord();
        i++;
    }

      while (binNodePtr)
      {
      int result;
```

```
            result = cmpPts(ptsBuffer,binNodePtr-
>graphPtr,LPoint.size());
        if (result == 0)
        {
            delete ptsBuffer;
                  return binNodePtr->heuristic;
        }
        else if (result == -1)
                  binNodePtr = binNodePtr->left;
        else
                  binNodePtr = binNodePtr->right;
      }
    delete ptsBuffer;
      return 0;
}

//*****************************************************************
*************

void StoreSubGraphHeuristic(GRAPH<point,int>& G,point &p,float
heuristic)
{
      int numOfNodes = G.number_of_nodes();

      float *ptsBuffer = new float[LPoint.size() * 2];
    if (!ptsBuffer)
      {
            cerr << endl << "NO MEMORY FOR POINT BUFFER (28)" << endl
;
        exit(1);
    }

      point x;

    int i = 0;
    forall(x,LPoint)
    {
        *(ptsBuffer + i) = x.xcoord();
        i++;
        *(ptsBuffer + i) = x.ycoord();
        i++;
    }

      struct binNode *binNodePtr;

    binNodePtr = (subGraphLoopUpTablePtr + numOfNodes)->binNodePtr;

    if (binNodePtr == NULL)
    {
      binNodePtr = (subGraphLoopUpTablePtr + numOfNodes)->binNodePtr
= new binNode;
          if (!binNodePtr)
            {
                  cerr << endl << "NO MEMORY FOR BIN NODE (29)" <<
endl ;
              exit(1);
      }
            binNodePtr->left = NULL;
        binNodePtr->right = NULL;
            binNodePtr->heuristic = heuristic;
            binNodePtr->graphPtr = ptsBuffer;
```

```
    }
    else

StoreSubGraphUpdatedHeuristic(binNodePtr,heuristic,ptsBuffer);
}


void StoreSubGraphHeuristic_2(GRAPH<point,int>& G,point &p,float
heuristic)
{
      int numOfNodes = G.number_of_nodes();

      float *ptsBuffer = new float[LPoint.size() * 2];
    if (!ptsBuffer)
      {
            cerr << endl << "NO MEMORY FOR POINT BUFFER (30)" << endl
;
        exit(1);
    }

      point x;

    int i = 0;
    forall(x,LPoint)
    {
        *(ptsBuffer + i) = x.xcoord();
        i++;
        *(ptsBuffer + i) = x.ycoord();
        i++;
    }
    *(ptsBuffer + i) = p.xcoord();
    i++;
    *(ptsBuffer + i) = p.ycoord();
    i++;

      struct binNode *binNodePtr;

    binNodePtr = (subGraphLoopUpTablePtr + numOfNodes)->binNodePtr;

    if (binNodePtr == NULL)
    {
      binNodePtr = (subGraphLoopUpTablePtr + numOfNodes)->binNodePtr
= new binNode;
          if (!binNodePtr)
            {
                  cerr << endl << "NO MEMORY FOR BIN NODE (31)" <<
endl ;
              exit(1);
       }
            binNodePtr->left = NULL;
        binNodePtr->right = NULL;
            binNodePtr->heuristic = heuristic;
            binNodePtr->graphPtr = ptsBuffer;
    }
    else

StoreSubGraphUpdatedHeuristic(binNodePtr,heuristic,ptsBuffer);
}

void StoreSubGraphUpdatedHeuristic(struct binNode *binNodePtr,float
heuristic,float *ptsBuffer)
```

```
{
    struct binNode *tempPtr = binNodePtr;
      int result;

      while (binNodePtr)
      {
            result = cmpPts(ptsBuffer,binNodePtr-
>graphPtr,LPoint.size());
        tempPtr = binNodePtr;
        if (result == 0)
        {
                if (binNodePtr->heuristic < heuristic)
                      binNodePtr->heuristic = heuristic;
                  return;
        }
        else if (result == -1)
                  binNodePtr = binNodePtr->left;
        else
                  binNodePtr = binNodePtr->right;
      }

      binNodePtr = new binNode;
    if (!binNodePtr)
      {
            cerr << endl << "NO MEMORY FOR BIN NODE (32)" << endl ;
        exit(1);
      }
      binNodePtr->left = NULL;
    binNodePtr->right = NULL;
      binNodePtr->heuristic = heuristic;
      binNodePtr->graphPtr = ptsBuffer;
    if (result == -1)
      tempPtr->left = binNodePtr;
    else
      tempPtr->right = binNodePtr;
      return;
}


float getSubGraphUpdatedHeuristic(GRAPH<point,int>& G, point&
endPoint)
{

      int numOfNodes = G.number_of_nodes();
      struct binNode *binNodePtr;

    binNodePtr = (subGraphLoopUpTablePtr + numOfNodes)->binNodePtr;

    if (binNodePtr == NULL)
            return 0;

      float *ptsBuffer = new float[LPoint.size() * 2 + 2];
    if (!ptsBuffer)
      {
            cerr << endl << "NO MEMORY FOR POINT BUFFER (33)" << endl
;
        exit(1);
    }

      point x;
```

```
    int i = 0;
    forall(x,LPoint)
    {
        *(ptsBuffer + i) = x.xcoord();
        i++;
        *(ptsBuffer + i) = x.ycoord();
        i++;
    }
    *(ptsBuffer + i) = endPoint.xcoord();
    i++;
    *(ptsBuffer + i) = endPoint.ycoord();
    i++;

      while (binNodePtr)
      {
      int result;
            result = cmpPts(ptsBuffer,binNodePtr-
>graphPtr,LPoint.size()+1);
        if (result == 0)
        {
            delete ptsBuffer;
                  return binNodePtr->heuristic;
        }
        else if (result == -1)
                  binNodePtr = binNodePtr->left;
        else
                  binNodePtr = binNodePtr->right;
      }
    delete ptsBuffer;
      return 0;
}

#endif

float redLineLength(void)
{
      float tempDistance = 0.0;

      if (LPoint.size() > 1)
    {
            int i,
            j ;
        point x1,
              x2;
        list_item listPtr;

            j =  LPoint.size();

        x1 = LPoint.front();
        listPtr = LPoint.first();
        for ( i = 1 ; i < j ; i++)
        {
            listPtr = LPoint.succ(listPtr);
            x2 = LPoint.contents(listPtr);
            tempDistance+= x1.distance(x2);
            x1 = x2;
        }
    }

    return tempDistance;
 }
```

```
int  cmpPts(float *p1, float *p2, int count)
{
      int i;
    count = count*2;
    for (i = 0 ; i < count ; i++,p1++,p2++)
      if ( *p1 < *p2)
            return -1;
        else if (*p1 > *p2)
            return 1;
      return 0;
}



void processArguments(int argc,char *argv[])
{
      char *cptr;

    if (argc == 1)
    {
            displayUsage();
            exit(0);
    }


      cptr = argv[1];

    if (strlen(argv[1]) == 1)
    {
            fprintf(stderr,"Invalid first argument\n");
            displayUsage();
            exit(0);
    }


    if (*cptr != '-')
    {
            fprintf(stderr,"Expecting a '-' in front of an
option\n");
            displayUsage();
            exit(0);
    }

    int i;

#ifndef FAST_CODE
    int counter = 0;
    bool unknownFlag = false;

    for ( i = 1 ; i < (int)strlen(argv[1]) ; i++)
    {
// graphic flag
            if (*(cptr + i ) == 'g')
        {
                  if (graphSelectedFlag)
            {
                        fprintf(stderr,"The 'g' option has been
repeated or 'G' is also specified\n");
                        displayUsage();
```

```
                exit(0);
            }
            counter++;
            graphSelectedFlag = true;
                  graphicFlag = true;
            continue;
        }


            if (*(cptr + i ) == 'G')
        {
                  if (graphSelectedFlag)
            {
                        fprintf(stderr,"The 'G' option has been
repeated or 'g' is also specified\n");
                        displayUsage();
                exit(0);
            }
            counter++;
            graphSelectedFlag = true;
                  endOfRunGraph = true;
            continue;
        }

// Step flag
            if (*(cptr + i ) == 's')
        {
                  if (stepFlag)
            {
                        fprintf(stderr,"The 's' option has been
repeated\n");
                        displayUsage();
                exit(0);
            }
            counter++;
                  stepFlag = true;
            continue;
        }

// Statistics flag
            if (*(cptr + i ) == 'S')
        {
                  if (statisticsFlag)
            {
                        fprintf(stderr,"The 'S' option has been
repeated\n");
                        displayUsage();
                exit(0);
            }
            counter++;
                  statisticsFlag = true;
            continue;
        }


// Display Stepped Output flag
            if (*(cptr + i ) == 'o')
        {
                  if (displayOutputFlag)
            {
```

```
                        fprintf(stderr,"The 'o' option has been
repeated\n");
                        displayUsage();
                    exit(0);
                }
                counter++;
                      displayOutputFlag = true;
                continue;
            }

                unknownFlag = true;
        }

        if (unknownFlag && counter)
        {
                fprintf(stderr,"Unknown argument entered in first
parameter\n");
                displayUsage();
                exit(0);
        }


        if (strlen(argv[1]) > 2 && unknownFlag)
        {
                fprintf(stderr,"Unknown arguments entered in first first
parameter\n");
                displayUsage();
                exit(0);
        }

#endif

#ifndef FAST_CODE
    char *validOptions = "rftTpOdb";
#else
    char *validOptions = "rftTpb";
#endif
// r - random points
// f - input file
// t - threshold
// T - alternate threshold
// p - point
// O - output file
// d - delay

      int startIndex;

#ifndef FAST_CODE
    if (counter)
      startIndex = 2;
    else
#endif
      startIndex = 1;

    while (startIndex != argc)
    {
          cptr = argv[startIndex];
        bool found = false;

          for (i = 0 ; i < (int)strlen(validOptions) ; i++)
        {
```

```
                if (*(cptr+1) == validOptions[i])
            {
                        found = true;
                break;
            }
        }

            if (!found)
          {
                fprintf(stderr,"Unknown argument in parameter
%d\n",startIndex);
              displayUsage();
                exit(0);
      }

          if (argc < startIndex +1)
          {
                fprintf(stderr,"Invalid number of parameters
provided\n");
                displayUsage();
                exit(0);
      }


          if (strlen(cptr) != 2 || *cptr != '-')
      {
                fprintf(stderr,"Expecting a '-' in front of
parameter %d\n",startIndex);
                displayUsage();
                exit(0);
          }

        if( *(cptr + 1) == 't')
        {
                if (thresholdSelectedFlag)
            {
                        fprintf(stderr,"The 't' option has been
repeated or 'T' is also specified\n");
                        displayUsage();
                exit(0);
            }
                initThreshold = currentThreshold = (float)atof(
argv[startIndex+1]);
            startIndex+=2;
                thresholdSelectedFlag = true;

        }
        if( *(cptr + 1) == 'T')
        {
                if (thresholdSelectedFlag)
            {
                        fprintf(stderr,"The 'T' option has been
repeated or 't' is also specified\n");
                        displayUsage();
                exit(0);
            }
                initThreshold = currentThreshold = (float)atof(
argv[startIndex+1]);
            altThresholdFlag = true;
                thresholdSelectedFlag = true;
            startIndex+=2;
```

```
        }
      else if (*(cptr +1) == 'p')
        {
                  startPoint = atoi( argv[startIndex+1]);
            if (startPoint < 1)
            {
                        fprintf(stderr,"Invalid start point entered:
%s\n",argv[startIndex+1]);
                        displayUsage();
                        exit(0);
                }
            startIndex+=2;
        }
      else if (*(cptr +1) == 'b')
        {
                  int binSizeValue = atoi( argv[startIndex+1]);
            if (binSizeValue < 1 || binSizeValue > 3)
            {
                        fprintf(stderr,"Invalid bin size (Enter 1,2
or 3)\n");
                        displayUsage();
                        exit(0);
                }
            if (binSizeValue == 1)
                        bin_size = 1000;
            else if (binSizeValue == 2)
                        bin_size = 10000;
            else
                        bin_size = 100000;
            startIndex+=2;
        }

#ifndef FAST_CODE
      else if (*(cptr +1) == 'd')
        {
                  delayGap = atoi( argv[startIndex+1]);
            if (delayGap < 1)
            {
                        fprintf(stderr,"Invalid delay entered:
%s\n",argv[startIndex+1]);
                        displayUsage();
                        exit(0);
                }
            if (!graphicFlag)
            {
                        fprintf(stderr,"Delay parameter requires the
selection of Graphics option\n");
                        displayUsage();
                        exit(0);
                }
            startIndex+=2;
        }
      else if (*(cptr +1) == 'O')
        {
                  outputFile = argv[startIndex+1];
                  FILE *fp;
            if ((fp = fopen(outputFile,"w")) == NULL)
            {
                        fprintf(stderr,"Unable to open output file
%s\n",outputFile);
                        displayUsage();
```

```
                        exit(0);
                    }
                fclose(fp);
                startIndex+=2;
            }
#endif

        else if (*(cptr +1) == 'f')
                {
                    if (argc < startIndex +2 )
                {
                        fprintf(stderr,"Expecting a filename as last
parameter - none supplied\n");
                        displayUsage();
                        exit(0);
                }
                    else if (argc > startIndex +2 )
                    {
                        fprintf(stderr,"Too many parameters supplied
- must be last set of parameter\n");
                        displayUsage();
                        exit(0);
                    }

                      filePointFlag = true;
              cptr = argv[startIndex + 1];
                readInGraph(cptr);
                startIndex+=2;

              }
         else if (*(cptr +1) == 'r')
             {
                    if (argc < startIndex +7 )
                {
                        fprintf(stderr,"Invalid number of parameters
supplied for generating random points\n");
                        displayUsage();
                        exit(0);
                }

                    else if (argc > startIndex +7 )
                    {
                        fprintf(stderr,"Invalid number of parameters
supplied for generating random points\nmust be last set of
parameters\n");
                        displayUsage();
                        exit(0);
                    }
                      randomPointFlag = true;

                  int points,
                xlow,
                    ylow,
                      xhigh,
                      yhigh,
                  prec;
                    points = atoi(argv[startIndex + 1]);
                    xlow = atoi(argv[startIndex + 2]);
                    ylow = atoi(argv[startIndex + 3]);
                    xhigh = atoi(argv[startIndex + 4]);
                    yhigh = atoi(argv[startIndex + 5]);
```

```
                prec = atoi(argv[startIndex + 6]);
              generateRandomGraph(points,xlow,ylow,xhigh,yhigh,prec);
            startIndex+=7;
      }
    }

    if (startPoint > numpoints)
      {
            fprintf(stderr,"Invalid start point entered: %d ( >
number of points in graph: %d) \n",startPoint,numpoints);
            displayUsage();
            exit(0);
    }

    if (!randomPointFlag && !filePointFlag)
    {
            fprintf(stderr,"No data source specified - either file or
random point generation\n");
            displayUsage();
            exit(0);
    }
}



void readInGraph(char *fileName)
{
      FILE *fp;

    fp = fopen(fileName,"r");

    if (fp == NULL)
    {
            fprintf(stderr,"File %s does not exist\n", fileName);
            displayUsage();
            exit(0);
    }

      fscanf(fp,"%d",&numpoints);

      point p;

    int i;

    for (i=0; i<numpoints ;i++)
    {
            float a,b;
        int count;

        count = fscanf(fp,"%f %f",&a,&b);

        if (count != 2)
        {
                  fprintf(stderr,"Invalided data in file\n");
                  displayUsage();
            exit(0);
        }

        if (i == 0)
        {
            maxX=minX=a;
```

```
            maxY=minY=b;
            }
        else
        {
                maxX=max(maxX,a);
                minX=min(minX,a);
                maxY=max(maxY,b);
                minY=min(minY,b);
        }
        p  = point(a,b);
            L.append(p);
    }
    fclose(fp);
}



void generateRandomGraph(int points,int xlow,int ylow,int xhigh,int
yhigh,int prec)
{
      numpoints = points;
    int i;
//    randomize(); // removed by diniz

      srand(time(NULL));
    float *xPtr, *yPtr;

    xPtr = (float *) malloc(numpoints *sizeof(float));
    yPtr = (float *) malloc(numpoints *sizeof(float));

    if (!(xPtr && yPtr))
    {
            fprintf(stderr,"No memory left to generate random
points\n");
        exit(0);
    }

      point p;
    int mult = 1;
    for (i = 0 ; i < prec ; i++)
      mult *= 10;

    for (i = 0 ; i < numpoints ; i++)
    {
      int loop = 1;

            while (loop)
        {
            int ii;
                  float j,k,l;

                  j = rand()%(xhigh - xlow);

              k = rand();

          while ((l = rand()) == 0)
            ;

            *(xPtr+i) =  xlow + j +
((float)((int)((k/l)*mult)%mult)/mult);
```

```
                    j = rand()%(yhigh - ylow);
                k = rand();

            while ((l = rand()) == 0)
              ;

              *(yPtr+i) =  ylow + j +
((double)((int)((k/l)*mult)%mult)/mult);

              if ( *(yPtr+i)> yhigh || *(xPtr + i ) > xhigh)
                    continue;

              loop = 0;
              for (ii = 0 ; ii < i ; ii++)
                          if ( *(xPtr + i) == *(xPtr + ii) && *(yPtr +
i) == *(yPtr + ii))
                  {
                                  loop = 1;
                      break;
                  }
          }

          if (i == 0)
          {
              maxX=minX=*(xPtr + i);
              maxY=minY=*(yPtr + i);
              }
          else
          {
                    maxX=max(maxX,*(xPtr + i));
                    minX=min(minX,*(xPtr + i));
                    maxY=max(maxY,*(yPtr + i));
                    minY=min(minY,*(yPtr + i));
          }

          p  = point(*(xPtr + i),*(yPtr + i));
              L.append(p);
      }
      FILE *fp;

      fp = fopen("random.dat","w");

      if (fp)
      {
              fprintf(fp,"%d\n",numpoints);

            for (i = 0 ; i < numpoints ; i++)
                    fprintf(fp,"%.*f %.*f\n",prec,*(xPtr +
i),prec,*(yPtr + i));
          fclose(fp);
        }
}


void displayUsage(void)
{
#ifdef FAST_CODE
        fprintf(stderr,"Usage: FAST VERSION \n");
        fprintf(stderr,"%s [<-p|t|T|b value>....] <-f value>|<-r
values>\n",progname);
#else
```

```
      fprintf(stderr,"Usage:\n");
      fprintf(stderr,"%s [<-gGsSo>] [<-p|t|T|b|O|d value>....] <-f
value>|<-r values>\n",progname);
      fprintf(stderr,"-g: show in graphic step mode\n");
      fprintf(stderr,"-G: show graphic at program termination\n");
      fprintf(stderr,"-s: step through in text mode\n");
      fprintf(stderr,"-S: calculate and display statistics about the
current problem\n");
      fprintf(stderr,"-o: display step output to screen\n");
#endif
      fprintf(stderr,"-p value : select a start point for the
processing [1.. number of graph points]\n");
      fprintf(stderr,"-t value : use a numerical threshold value in
this calculation\n");
      fprintf(stderr,"-T value : use a percentage threshold value in
this calculation\n");
      fprintf(stderr,"-b value : use to change binsize 1 - 1000, 2 -
10000 or 3 - 100000\n");
#ifndef FAST_CODE
      fprintf(stderr,"-d value : delay (in seconds) between screen
updates in auto graphics mode\n");
      fprintf(stderr,"-O value : file name of file were output is
sent\n");
#endif
      fprintf(stderr,"-f value : file name of input data file\n");
      fprintf(stderr,"-r values : random graph generation 6 values
expected\n");
      fprintf(stderr,"            numPoints - number of points\n");
      fprintf(stderr,"            xlow - lowest x value limit of
random points\n");
      fprintf(stderr,"            ylow - lowest y value limit of
random points\n");
      fprintf(stderr,"            xhigh - highest x value limit of
random points\n");
      fprintf(stderr,"            yhigh - highest y value limit of
random points\n");
      fprintf(stderr,"            precision - number of decimal
places in point\n");
      fprintf(stderr,"Note: random point problem written to a file -
random.dat");
}
```

APPENDIX B: SUMMARISED SEARCH RESULTS FOR EXAMPLES 2 AND 3 IN CHAPTER 5 USING THE RESTRICTIVE SEARCH APPROACH

Table B.1: Summarised search process of Example 2 - BURMA14 using the restrictive SLA*-TSP approach

	Level													
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	(5,5)	(5,4,5)												
	20.1060	22.6717												
2	(5,5)	(5,4,5)	(5,4,3,5)											
	22.6717	22.6717	25.2374											
3	((,5)*	(5,4,5)	(5,4,3,5)	(5,4,3,11,5)										
	25.2374	25.2374	25.2374	25.2386										
4	(5,5)*	(5,4,5)	(5,4,3,5)	(5,4,3,11,5)	(5,4,3,11,10,5)	(5,4,3,11,10,14,5)								
	25.2386	25.2386	25.2386	25.2386	25.0096	28.6828								
5	(5,5)*	(5,4,5)	(5,4,2,5)	(5,4,2,3,5)										
	26.1998	26.1998	26.1998	26.8788										
6	(5,5)*	(5,4,5)	(5,4,3,5)	(5,4,3,2,5)	(5,4,3,2,11,5)	(5,4,3,2,11,10,5)	(5,4,3,2,11,10,13,5)							
	26.4773	26.4773	26.4773	26.4773	26.3222	24.2855	27.5802							
7	(5,5)*	(5,4,5)	(5,4,2,5)	(5,4,2,3,5)	(5,4,2,3,11,5)									
	26.8788	26.8788	26.8788	26.8788	27.0832									
8	(5,5)*	(5,4,5)	(5,4,2,5)	(5,4,2,3,5)										
	27.2281	27.2281	27.2281	27.9818										
9	(5,5)*	(5,4,5)	(5,4,3,5)	(5,4,3,2,5)	(5,4,3,2,11,5)	(5,4,3,2,11,10,5)	(5,4,3,2,11,10,13,5)	(5,4,3,2,11,10,13,9,5)						
	27.5802	27.5802	27.5802	27.5802	27.5802	27.5802	27.5802	27.6467						
10	(5,5)*	(5,4,5)	(5,4,3,5)	(5,4,3,2,5)	(5,4,3,2,11,5)	(5,4,3,2,11,10,5)	(5,4,3,2,11,10,13,5)	(5,4,3,2,11,10,13,9,5)	(5,4,3,2,11,10,13,9,12,5)					
	27.6467	27.6467	27.6467	27.6467	27.6467	27.6467	27.6467	27.6467	29.8966					
11	(5,5)*	(5,4,5)	(5,4,3,5)	(5,4,3,2,5)	(5,4,3,2,11,5)	(5,4,3,2,11,10,5)	(5,4,3,2,11,10,14,5)	(5,4,3,2,11,10,14,12,5)						
	27.9588	27.9588	27.9588	27.9588	27.9588	27.9588	27.9588	30.9811						
12	(5,5)*	(5,4,5)	(5,4,2,5)	(5,4,2,3,5)	(5,4,2,3,11,5)	(5,4,2,3,11,10,5)	(5,4,2,3,11,10,13,5)	(5,4,2,3,11,10,13,9,5)						

	Level													
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	27.9817	27.9817	27.9817	27.9817	27.0832	27.0832	27.0832	27.1497						
13	(5,5)*	(5,4,5)	(5,4,2,5	(5,4,2,3,5)	(5,4,2,3,11,5)	(5,4,2,3,11,10,5)	(5,4,2,3,11,10,13,5)	(5,4,2,3,11,10,13,9,5)	(5,4,2,3,11,10,13,9,12,5)					
	28.0482	28.0482	28.0482	28.0482	28.0482	28.0482	27.1497	27.1497	29.3996					
14	(5,5)*	(5,4,5)	(5,4,3,5)	(5,4,3,2,5)	(5,4,3,2,11,5)	(5,4,3,2,11,10,5)	(5,4,3,2,11,10,13,5)	(5,4,3,2,11,10,13,9,5)						
	28.5520	28.5520	28.5520	28.5520	28.5520	28.5520	28.5520	29.8966						
15	(5,5)*	(5,4,5)	(5,4,3,5)	(5,4,3,11,5)	(5,4,3,11,10,5)	(5,4,3,11,10,14,5)	(5,4,3,11,10,14,12,5)							
	28.6828	28.6828	28.6828	28.6828	28.6828	28.6828	31.7052							
16	(5,5)*	(5,4,5)	(5,4,2,5)	(5,4,2,3,5)	(5,4,2,3,11,5)	(5,4,2,3,11,10,5)	(5,4,2,3,11,10,13,5)	(5,4,2,3,11,10,13,9,5)						
	28.9536	28.9536	28.9536	28.9536	28.0550	28.0550	28.0550	29.3996						
17	(5,5)*	(5,4,5)	(5,4,3,5)	(5,4,3,11,5)	(5,4,3,11,13,5)	(5,4,3,11,13,9,5)								
	29.0942	29.0942	29.0942	29.0942	29.0942	29.1836								
18	(5,5)*	(5,4,5)	(5,4,3,5)	(5,4,3,11,5)	(5,4,3,11,13,5)	(5,4,3,11,13,9,5)	(5,4,3,11,13,9,12,5)							
	29.1836	29.1836	29.1836	29.1836	29.1836	29.1836	31.4335							
19	(5,5)*	(5,4,5)	(5,4,3,5)	(5,4,3,11,5)	(5,4,3,11,10,5)	(5,4,3,11,10,13,5)	(5,4,3,11,10,13,9,5)							
	29.3672	29.3672	29.3672	29.3672	29.3672	29.3672	29.9363							
20	(5,5)*	(5,4,5)	(5,4,3,5)	(5,4,3,2,5)	(5,4,3,2,11,5)	(5,4,3,2,11,10,5)	(5,4,3,2,11,10,13,5)	(5,4,3,2,11,10,13,9,5)	(5,4,3,2,11,10,13,9,12,5)	(5,4,3,2,11,10,13,9,12,5)	(5,4,3,2,11,10,13,9,12,8,5)			
	29.8966	29.8966	29.8966	29.8966	29.8966	29.8966	29.8966	29.8966	29.8966	29.8966	30.0712			
21	(5,5)*	(5,4,5)	(5,4,3,5)	(5,4,3,11,5)	(5,4,3,11,10,5)	(5,4,3,11,10,13,5)	(5,4,3,11,10,13,9,5)	(5,4,3,11,10,13,9,12,5)						
	29.9363	29.9363	29.9363	29.9363	29.9363	29.9363	29.9363	32.1862						
22	(5,5)*	(5,4,5)	(5,4,3,5)	(5,4,3,11,5)	(5,4,3,11,13,5)									
	30.0660	30.0660	30.0660	29.3672	30.3390									
23	(5,5)*	(5,4,5)	(5,4,3,5)	(5,4,3,2,5)	(5,4,3,2,11,5)	(5,4,3,2,11,10,5)	(5,4,3,2,11,10,13,5)	(5,4,3,2,11,10,13,9,5)	(5,4,3,2,11,10,13,9,12,5)	(5,4,3,2,11,10,13,9,12,5)	(5,4,3,2,11,10,13,9,12,8,5)	(5,4,3,2,11,10,13,9,12,8,1,5)		
	30.0712	30.0712	30.0712	30.0712	30.0712	30.0712	30.0712	30.0712	30.0712	30.0712	30.0712	32.1502		
24	(5,5)*	(5,4,5)	(5,4,2,5)	(5,4,2,3,5)	(5,4,2,3,11,5)	(5,4,2,3,11,10,5)	(5,4,2,3,11,10,13,5)	(5,4,2,3,11,10,13,9,5)	(5,4,2,3,11,10,13,9,12,5)	(5,4,2,3,11,10,13,9,12,8,5)				

	Level													
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	30.2982	30.2982	30.2982	29.3996	29.3996	29.3996	29.3996	29.3996	29.3996	29.5742				
25	(5,5)*	(5,4,5)	(5,4,3,5)	(5,4,3,11,5)	(5,4,3,11,10,5)	(5,4,3,11,10,13,5)	(5,4,3,11,10,13,9,5)							
	30.3753	30.3753	30.3753	30.3753	30.3390	30.3390	32.1862							
26	(5,5)*	(5,4,5)	(5,4,3,5)	(5,4,3,2,5)	(5,4,3,2,11,5)	(5,4,3,2,11,10,5)	(5,4,3,2,11,10,13,5)	(5,4,3,2,11,10,13,9,5)	(5,4,3,2,11,10,13,9,12,5)					
	30.4113	30.4113	30.4113	30.4113	30.4113	30.4113	30.4113	30.4113	32.1502					
27	(5,5)*	(5,4,5)	(5,4,2,5)	(5,4,2,3,5)	(5,4,2,3,11,5)	(5,4,2,3,11,10,5)	(5,4,2,3,11,10,13,5)	(5,4,2,3,11,10,13,9,5)	(5,4,2,3,11,10,13,9,12,5)	(5,4,2,3,11,10,13,9,12,8,5)	(5,4,2,3,11,10,13,9,12,8,1,5)			
	30.4727	30.4727	30.4727	30.4727	29.5742	29.5742	29.5742	29.5742	29.5742	29.5742	31.6532			
28	(5,5)*	(5,4,5)	(5,4,2,5)	(5,4,2,3,5)	(5,4,2,3,11,5)	(5,4,2,3,11,10,5)	(5,4,2,3,11,10,13,5)	(5,4,2,3,11,10,13,9,5)	(5,4,2,3,11,10,13,9,8,5)	(5,4,2,3,11,10,13,9,8,1,5)				
	30.8128	30.8128	30.8128	30.8128	29.9143	29.9143	29.9143	29.9143	29.9143	32.7332				
29	(5,5)*	(5,4,5)	(5,4,3,5)	(5,4,3,2,5)	(5,4,3,2,11,5)	(5,4,3,2,11,10,5)	(5,4,3,2,11,10,14,5)	(5,4,3,2,11,10,14,12,5)	(5,4,3,2,11,10,14,12,7,5)					
	30.9811	30.9811	30.9811	30.9811	30.9811	30.9811	30.9811	30.9811	33.2453					
30	(5,5)*	(5,4,5)	(5,4,3,5)	(5,4,3,2,5)	(5,4,3,2,11,5)	(5,4,3,2,11,10,5)	(5,4,3,2,11,10,14,5)	(5,4,3,2,11,10,14,9,5)	(5,4,3,2,11,10,14,9,12,5)					
	31.3685	31.3685	31.3685	31.3685	31.3685	31.3685	31.3685	31.3685	33.4149					
31	(5,5)*	(5,4,5)	(5,4,3,5)	(5,4,3,11,5)	(5,4,3,11,13,5)	(5,4,3,11,13,9,5)	(5,4,3,11,13,9,12,5)	(5,4,3,11,13,9,12,8,5)						
	31.4335	31.4335	31.4335	31.4335	31.4335	31.4335	31.4335	31.6081						
32	(5,5)*	(5,4,5)	(5,4,3,5)	(5,4,3,2,5)	(5,4,3,2,11,5)	(5,4,3,2,11,10,5)	(5,4,3,2,11,10,13,5)	(5,4,3,2,11,10,13,9,5)						
	31.5743	31.5743	31.5743	31.5743	31.5743	31.5743	31.5743	32.1502						
33	(5,5)*	(5,4,5)	(5,4,2,5)	(5,4,2,3,5)	(5,4,2,3,11,5)	(5,4,2,3,11,10,5)	(5,4,2,3,11,10,13,5)	(5,4,2,3,11,10,13,9,5)	(5,4,2,3,11,10,13,9,12,5)	(5,4,2,3,11,10,13,9,12,8,5)	(5,4,2,3,11,10,13,9,12,8,1,5)	(5,4,2,3,11,10,13,9,12,8,1,7,5)	(5,4,2,3,11,10,13,9,12,8,1,7,14,5)	(5,4,2,3,11,10,13,9,12,8,1,7,14,6,5)
	32.5517	32.5517	32.5517	32.5517	31.6532	31.6532	31.6532	31.6532	31.6532	31.6532	31.6532	30.8276	29.0084	31.4850
34	(5,5)	(5,4,5)	(5,4,2,5)	(5,4,2,3,5)	(5,4,2,3,11,5)	(5,4,2,3,11,10,5)	(5,4,2,3,11,10,13,5)	(5,4,2,3,11,10,13,9,5)	(5,4,2,3,11,10,13,9,12,5)	(5,4,2,3,11,10,13,9,12,8,5)	(5,4,2,3,11,10,13,9,12,8,1,5)	(5,4,2,3,11,10,13,9,12,8,1,7,5)	(5,4,2,3,11,10,13,9,12,8,1,7,6,5)*	(5,4,2,3,11,10,13,9,12,8,1,7,6,14,5)
	32.5517	32.5517	32.5517	32.5517	31.6532	31.6532	31.6532	31.6532	31.6532	31.6532	31.6532	30.8276	29.3352	30.8785
35	(5,5)	(5,4,5)	(5,4,2,5)	(5,4,2,3,5)	(5,4,2,3,11,5)	(5,4,2,3,11,10,5)	(5,4,2,3,11,10,13,5)	(5,4,2,3,11,10,13,9,5)	(5,4,2,3,11,10,13,9,12,5)	(5,4,2,3,11,10,13,9,12,8,5)	(5,4,2,3,11,10,13,9,12,8,1,5)	(5,4,2,3,11,10,13,9,12,8,1,7,5)*	(5,4,2,3,11,10,13,9,12,8,1,7,6,5)	(5,4,2,3,11,10,13,9,12,8,1,7,6,14,5)
	32.5517	32.5517	32.5517	32.5517	31.6532	31.6532	31.6532	31.6532	31.6532	31.6532	31.6532	30.8785	30.8785	30.8785

Table B.2: Summarised search process of Example 3 - 12-city problem using the restrictive SLA*-TSP approach

	Level											
	0	1	2	3	4	5	6	7	8	9	10	11
1	(8,8)	(8,9,8)										
	289.7910	306.5002										
2	(8,8)*	(8,9,8)	(8,9,7,8)	(8,9,7,11,8)								
	306.5002	306.5002	266.1971	378.1523								
3	(8,8)*	(8,7,8)	(8,7,11,8)									
	329.6518	329.6518	441.6061									
4	(8,8)*	(8,9,8)	(8,9,7,8)	(8,9,7,11,8)	(8,9,7,11,12,8)							
	378.1520	378.1520	378.1521	378.1521	381.7726							
5	(8,8)*	(8,9,8)	(8,9,7,8)	(8,9,7,12,8)	(8,9,7,12,10,8)							
	379.3450	379.3450	379.3450	379.3448	397.2194							
6	(8,8)*	(8,9,8)	(8,9,7,8)	(8,9,7,11,8)	(8,9,7,11,12,8)	(8,9,7,11,12,10,8)	(8,9,7,11,12,10,4,8)					
	381.7722	381.7722	381.7722	381.7722	381.7722	334.7411	416.7090					
7	(8,8)*	(8,9,8)	(8,9,7,8)	(8,9,7,11,8)	(8,9,7,11,10,8)	(8,9,7,11,10,4,8)						
	391.7951	391.7951	391.7951	391.7951	391.7951	485.9223						
8	(8,8)*	(8,9,8)	(8,9,7,8)	(8,9,7,12,8)	(8,9,7,12,10,8)	(8,9,7,12,10,4,8)						
	397.2196	397.2196	397.2196	397.2196	397.2196	491.3512						
9	(8,8)*	(8,9,8)	(8,9,3,8)	(8,9,3,1,8)								
	415.6662	415.6662	415.6662	431.8661								
10	(8,8)*	(8,9,8)	(8,9,7,8)	(8,9,7,11,8)	(8,9,7,11,12,8)	(8,9,7,11,12,10,8)	(8,9,7,11,12,10,4,8)	(8,9,7,11,12,10,4,5,8)				
	416.7090	416.7090	416.7090	416.7090	416.7090	416.7090	416.7090	421.4360				
11	(8,8)*	(8,9,8)	(8,9,7,8)	(8,9,7,11,8)	(8,9,7,11,12,8)	(8,9,7,11,12,10,8)	(8,9,7,11,12,10,4,8)	(8,9,7,11,12,10,4,5,8)	(8,9,7,11,12,10,4,5,6,8)	(8,9,7,11,12,10,4,5,6,2,8)		
	421.4360	421.4360	421.4360	421.4360	421.4360	421.4360	421.4360	421.4360	370.6454	375.5651		
12	(8,8)	(8,9,8)	(8,9,7,8)	(8,9,7,11,8)	(8,9,7,11,12,8)	(8,9,7,11,12,10,8)	(8,9,7,11,12,10,4,8)	(8,9,7,11,12,10,4,5,8)	(8,9,7,11,12,10,4,5,6,8)*	(8,9,7,11,12,10,4,5,6,2,8)	(8,9,7,11,12,10,4,5,6,2,3,8)	(8,9,7,11,12,10,4,5,6,2,3,1,8)
	421.4360	421.4360	421.4360	421.4360	421.4360	421.4360	421.4360	421.4360	375.5651	375.5651	363.5469	401.6782
13	(8,8)	(8,9,8)	(8,9,7,8)	(8,9,7,11,8)	(8,9,7,11,12,8)	(8,9,7,11,12,10,8)	(8,9,7,11,12,10,4,8)	(8,9,7,11,12,10,4,5,8)	(8,9,7,11,12,10,4,5,6,8)*	(8,9,7,11,12,10,4,5,6,2,8)	(8,9,7,11,12,10,4,5,6,2,1,8)	(8,9,7,11,12,10,4,5,6,2,1,3,8)
	421.4360	421.4360	421.4360	421.4360	421.4360	421.4360	421.4360	421.4360	384.6515	384.6514	384.6515	384.9909
14	(8,8)	(8,9,8)	(8,9,7,8)	(8,9,7,11,8)	(8,9,7,11,12,8)	(8,9,7,11,12,10,8)	(8,9,7,11,12,10,4,8)	(8,9,7,11,12,10,4,5,8)	(8,9,7,11,12,10,4,5,6,8)*	(8,9,7,11,12,10,4,5,6,2,8)	(8,9,7,11,12,10,4,5,6,2,1,8)	(8,9,7,11,12,10,4,5,6,2,1,3,8)
	421.4360	421.4360	421.4360	421.4360	421.4360	421.4360	421.4360	421.4360	384.9909	384.9909	384.9909	384.9909