MODIFIED MST ALGORITHMS

FOR TSP TOURS

By

XIAOZHEN WANG

Bachelor of Laws
East China Normal University
Shanghai, China
1982

Master of Laws
East China Normal University
Shanghai, China
1985

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 2000

MODIFIED MST ALGORITHMS

FOR TSP TOURS

Thesis Approved:

Jacques E. LaFrance

Thesis Adviser

J Chandler

[illegible]

Wayne B. Powell

Dean of the Graduate College

ACKNOWLEDGMENTS

This thesis develops some modifications of the MST algorithm for generating a TSP tour. The TSP (Traveling Salesman Problem) is a classical problem in combinatorial optimization. Generally the solution of the TSP is to generate the shortest cyclic tour which visits every vertex in a complete graph G exactly once and starts and ends at the same vertex. Since the TSP tour is a NP-hard problem, we can not solve it in the polynomial time. In the real world, there are many approximation algorithms those that are near-optimal solutions for generating the TSP tour in the polynomial time. Of these algorithms, the MST (Minimum Spanning Tree) algorithm is an early classic solution, but the quality of the result of the algorithm is seemed as not good enough for a practical purpose. On the other side, there are many heuristic methods, which can generate reasonably efficient results but do not guarantee feasible solutions and are used without a theoretical guarantee.

The subject of the thesis is to combine the MST algorithm with some efficient heuristic methods to innovate some modified variety of the MST algorithm in the TSP field. The study focused on the symmetric TSP that satisfying the triangle inequality criteria. Two modified algorithms are developed. One combines the MST algorithm with a modified variety of the edge exchange algorithm. The other combines the MST with a random insertion algorithm. Both the modifications of the MST algorithm are expected to improve the quality of the TSP tour with keeping the advantage of the MST algorithm.

I wish to express my sincere appreciation to Professors Jacques LaFrance, John P. Chandler and Jing Peng, who have served as members of my master committee. The special thanking goes to my major adviser, Dr. LaFrance, for his trust, concern and valuable guidance. I am also thankful to Dr. Chandler and Dr. Peng for their advice and encouragement.

My mother, my wife and my brothers deserve my deep appreciation for their encouragement, patience and financial support.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

TSP and MST

The Traveling Salesman Problem (TSP) is a classical problem in combinatorial optimization. A TSP is based on the assumption that there is a traveling salesman who wants to visit every city exactly once in a certain area, and end up at the city which he starts from. While doing so he also tries reduce cost and time by making the shortest trip possible. So the major task for the salesman is to find the so-called the "shortest Hamiltonian tour" in a complete graph G = (V, E). Here V represents the cities (vertices); and E represents the paths (edges) between different cities.

A Hamiltonian tour itself is a journey in a complete graph, which the subject visits every vertex exactly once through the path between any two vertices in the graph, and forms a closed cycle by starting and ending the tour in the same vertex.

The TSP has many variations and many formulations have been developed since it was first discussed in the eighteenth century. [Noschang 1997] This thesis will only focus on the symmetric TSP of G = (V, E) on a Euclidean plane which satisfies the triangular inequality. The symmetric TSP is the basic TSP formulation, which means that the weight of any path between any two vertices is always the same in the any two directions connecting the two vertices. On a Euclidean plane, the weight of the path is determined by the length. At the same time, there is an asymmetric TSP in which every path will be associated with different weights based on different directions. This thesis focuses mainly on the algorithms, which are used to generate an optimal Hamiltonian tour of the symmetric TSP.

The TSP is an important computational model. It is the basis of many real world problems, and it is also the concept that forms a central component of many real-world applications. [Noschang 1997] For example, wire distribution, network connection and design of printed circuit boards are the common cases related with the TSP theory and the associated algorithms.

But the TSP is also a NP-hard problem. That means we cannot solve it in polynomial time. Since the TSP is so important, we must find some efficient algorithms to solve the problem, and not just to provide a solution to its NP-hard property. Generally, there are three ways of finding a solution. Firstly, if the size (the number of vertices) of the problem were relatively small, an enumeration algorithm may be used to exhaust all possible tours for finding out the shortest tour. Secondly, we can design some exponential algorithms to solve the problem perfectly. Lastly, if the size of the problem were relatively large, the enumeration algorithm and the exponential algorithm will become infeasible. In this case, the feasible way we can go is to find some near-optimal solutions that can be implemented in polynomial time. These solutions belong to a class known as approximation algorithms. [Cormen, Leiserson, and Rivest 1995] Among approximation algorithms there is a subclass called heuristic methods. The heuristic methods simplify or educate the guess that reduces or limits the search for solutions.

Heuristics do not guarantee feasible solutions, and are often used with no theoretical support. In fact, the TSP is only one of the few major success cases for optimization techniques using heuristic methods. [Aarts and Lenstra 1997] Many approximation algorithms and heuristic methods have been developed to solve the TSP, and this is still an active area. The possibility of finding a better algorithm, or getting some better quality of result from existing algorithms, is still an open problem of major interest.

Of those algorithms concerning the TSP, the Minimum Spanning Tree (MST) tour is an early and classic resolution. The MST is the shortest path tree of a given graph G = (V, E) in which V is the set of vertices and E is the set of edges that connect two vertices. Although there may be many paths (edges) among the vertices, the MST is a tree that links all nodes (vertices) in the graph with least paths (V-1) and generates the shortest total length of the paths. Based on this quality of the MST, there is the MST algorithm frequently used to demonstrate the approximately optimal solution of the TSP. [Lawler, Lenstra, Kan and Shmoys 1985]

If the MST were a cyclical graph, it could have been a direct optimal solution of the symmetric TSP since it already connects every node only once, and the total length of the MST paths is the shortest or most cost-efficient. But a MST is not a cyclical graph. There is no way to generate a cyclical tour and visit every node (vertex) only once by directly walking through the all paths of the MST. For generating a Hamiltonian tour from a MST, we must add some complement methods or techniques to convert the MST to a cyclic tour. It means that for generating a TSP tour with the MST algorithm we need at least two basic steps. First, generating a MST to get a set of the shortest paths. Second, based on the MST, using some techniques to re-connect the paths of the MST and finally to construct a Hamiltonian tour.

Around the MST algorithm, there are several techniques to be devised to convert a MST to a near-optimal cyclical tour for solving the TSP. First of all, there are two similar algorithms. One is using the depth first traversal of the MST to construct the TSP tour [Lawler, Lenstra, Kan and Shmoys 1985], and the other is using a preorder walk of the MST to construct the TSP tour. [Cormen, Leiserson, and Rivest 1995] The complexity of the both methods is $O(n^2)$, while the quality of the resulted tour T is $T(n) <= 2OPT(n)$.

The Christofrides algorithm has different features from the above algorithms. This method requires a MST first for getting the set of vertices. Then it converts the MST to get a new connected graph G'. G' is an Euler graph with an Euler tour. By traversing the Euler tour, a TSP tour is generated. With this algorithm, constructing the MST needs $O(n^2)$, and constructing a TSP tour needs $O(n^3)$. But the quality of resulted T(n) is improved as $T(n) <= 3/2OPT(n)$. [Cormen, Leiserson, and Rivest 1995] [Aarts and Lenstra 1997] Generally the algorithms based on the MST are seen as the non-practical efficient algorithms to solve the TSP since either the worst case bound of the resulted tour is 2, or the time cost is relatively high, up to $O(n^3)$. [Cormen, Leiserson, and Rivest 1995]

Meanwhile, during the last several decades, there were relatively many heuristic methods invented to solve TSP tours. For example, the edge-exchange method (k-opt method) [Lin 1965] [Aarts and Lenstra 1997], the nearest neighbor method [Aarts and Lenstra 1997] [Cormen, Leiserson, and Rivest 1995], the insertion method [Dakin 1997], and the strip algorithm [Gu 1998]. Also, some general algorithms are adopted as the strategy for solving TSP, such as the dynamic programming algorithm [Smith 1987], the greedy algorithm [Aarts and Lenstra 1997] , and the branch-and-bound algorithm. Of these algorithms, some are efficient to get near optimal results under some conditions and constraints, but they are generally time-consuming. There is also a great weakness: we have no precise idea about how far the solutions are from the optimal results. [Noschang 1997]

From the achievement of heuristic methods, the basic idea of this research is that if we can combine some heuristic techniques with the MST algorithm, then we may be able to improve the result of the MST algorithm. This kind of thinking has led the author to look for some efficient methods to locate better solutions for the TSP.

Subject and Purpose

Based on the acknowledgment of the inefficiency of the MST algorithm when it constructs the TSP tour, the main purpose of this research is to find some more efficient techniques that can improve the result of the MST algorithm. Here the improvement means to generate a shorter TSP tour or to receive a more optimized solution than the original MST algorithm. Meanwhile, the improvement should not increase the time cost of total computing. For seeking these techniques, we need to turn our eyes to other heuristics that solved the TSP better than the MST algorithm. When we studied some efficient heuristics carefully, we got the main idea and the subject of this research. The idea is to try combining some techniques of some heuristics with the classical MST algorithm to improve the performance of the MST algorithm. Based on the idea, the subject of this research is to modify the MST algorithm with some technique combinations. Two new methods are devised by adding some techniques in the classical MST algorithm; these techniques are used to convert a MST to a TSP tour.

For implementing the purpose and the subject, there are two main tasks needed to carry out. The first is to design and experiment the two new algorithms. The second is to evaluate them by comparing them with the original MST algorithm and some selected heuristics, and to see if the results satisfying our expectation. Otherwise, for learning the new algorithms visually, some GUI (the graphic user interface) design will be done and the graphical results of the experiment will be shown on the screen.

The Frame of the Thesis

In Chapter 1, the basic concepts of the research are given. Then there is a concise discussion of literature review in Chapter 2. Chapter 3 is the detailed statements of the research tasks. Chapter 4 is the description of the design of the new algorithms.
In Chapter 5, we will discuss the experiment results of the new algorithms based on various comparisons. Finally we will represent our conclusions in Chapter 6.

CHAPTER II

LITERATURE SURVEY

History

Mathematicians have been interested in the traveling salesman problem since the 18th century. In 1759, there was a "knight's tour" problem discussed by both Euler and Vandermonde. [Noschang 1997] In 1856, Hamilton developed his "Iconsian Calculus" for graph problems marked as the Icosian Game. [Noschang 1997] In 1930, Menger mentioned the "messager problem" for finding the shortest Hamiltonian path. [Noschang 1997] But the term of "traveling salesman problem" may have been used at the first time in 1931 or 1932. [Lawler, Lenstra, Kan and Shmoys 1985]

Probably the main reason that the TSP is one of the most interesting problems of mathematicians and computer scientists is its genre: combinatorial optimization. [Lawler, Lenstra, Kan and Shmoys 1985] As a classical combinatorial optimization problem, the TSP served as an initial proving ground for optimization algorithms and heuristics, and became one of the major success stories for optimization. [Johnson and McGeoch 1995] Up to now, the TSP has developed into a class of its own. Many variants of the TSP have been researched: the asymmetric traveling salesman problem, the multi-salesmen problem, the graphical traveling salesman problem, the bottleneck traveling salesman problem, the prize collecting traveling salesman problem, the vehicle routing traveling salesman problem, the Euclidean traveling salesman selection problem, the circulant traveling salesman problem, the Chinese postman problem, the on-line traveling salesman problem, the rural postman problem, the maximum or minimum latency problem, the Eulerian tour problem, the Hamiltonian cycle traveling salesman problem, the bipartite traveling salesman problem, the remote traveling salesman problem, the geometric covering salesman problem, the tree and tour cover traveling salesman problem, the k-template traveling salesman problem, and some others. [Noschang 1997] [Lawler, Lenstra, Kan and Shmoys 1985] Of them, there is the symmetric traveling salesman problem, which is the classical TSP. [Noschang 1997]

The main purpose of the research of the classical TSP is to find an algorithm that can figure out an optimal tour (the shortest tour) for the salesman. But since there are (n-1)! possible tours for a single TSP with n number of cities, the computing time becomes a very important property of the algorithm. Up till now, the classical TSP is still considered to be a NP-hard problem. [Garey and Johnson 1979] That means it is impossible to find a real optimal algorithm which can solve the TSP in a time that increases polynomially with the size n of the problem. [Noschang 1997] Under these assumptions, the researchers of the TSP have two general alternatives: either to find out some heuristics that merely result in "near optimal" tours, or to develop some optimization algorithms that run in a practically acceptable time. [Johnson and McGeoch 1995] Many ideas and algorithms that are related to the combinatorial optimization have been developed to solve the TSP for last several decades.

Around 1954, Dantzig, Fulkerson and Johnson mentioned the resolutions to the

TSP with the linear programming concept. [Willard 1970]

In 1958 and the following years, Gomory invented several algorithms with the cutting planes and facets method to generate optimal TSP tours. His algorithms gave effects on the development of the integer programming. [Lawler, Lenstra, Kan and Shmoys 1985]

Then in 1960, Miller, Tucker and Zemlin provided their integer programming solutions of the TSP. [Miller, Tucker and Zemlin 1960] [Willard 1970]

In 1962, Bellman, also Held and Karp, introduced an enumerative optimization method the dynamic programming and applied it to solve the TSP. [Ghalib 1979] [Lawler, Lenstra, Kan and Shmoys 1985]

One year later, the so-called "branch and bound" algorithm was used to solve the TSP by Little, Murty, Sweeney and Karel. [Willard 1970] [Lawler, Lenstra, Kan and Shmoys 1985]

In 1965, Lin provided his edge-exchange method. It was further defined and developed in 1973 by Lin and Kernighan. [Lin 1965]] [Lawler, Lenstra, Kan and Shmoys 1985]

In 1970s, based on the broad discussion about NP concept, the focus of the TSP research was shifted to look for approximation algorithms and heuristic methods. [Lawler, Lenstra, Kan and Shmoys 1985] In 1975, Papadimitrion and Steiglitz discussed the complexity of the local search the heuristics for the TSP and gave a polynomial predictable evaluation. [Ghalib 1979] [Papadimitrion and Steiglitz 1977] In 1976, Christofides, based on the MST algorithm, presented his elegant heuristic that gave the best result in the worst-case analysis of the TSP resolution up to now.

In the two recent decades, the research interesting is generally focused on heuristics. Many near-optimal methods have been innovated such as the neighborhood search, the simulated annealing, the Tabu search, the genetic algorithm, the neural network, the clustering heuristics, the ant systems, the mimetic algorithm and space-filling heuristic. [Noschang 1997] [Lawler, Lenstra, Kan and Shmoys 1985]

Algorithm Review

Of all algorithms and heuristics related to the TSP, the MST algorithm is a classical polynomial resolution for the TSP. Since every TSP tour is a spanning tree with an extra edge approaching the starting vertex the second time, we can generate a TSP tour from a MST based on the triangular inequality criteria. [Lawler, Lenstra, Kan and Shmoys 1985]

Around the MST, there are several techniques to be devised to convert a MST to a near-optimal TSP tour. One method applies the depth-first traversal of the MST. Starting at an arbitrary leaf of the minimum spanning tree, the depth-first traversal will traverse every edge of the tree exactly twice. Based on the double path of the MST, a TSP tour can be generated by using some "shortcuts" to replace some edges and to avoid visiting a vertex twice. [Lawler, Lenstra, Kan and Shmoys 1985] The other method uses a preordered walk of the MST to generate a cycle tour. [Cormen, Leiserson, and Rivest 1995] Both algorithms are based on a double path of the MST. By walking along the double path, a cyclic TSP tour can be generated. These techniques are called the MST algorithm, or the double MST algorithm. [Lawler, Lenstra, Kan and Shmoys 1985] The

MST algorithm gives the lower bound of the optimal TSP tour immediately since the MST is always shorter than the real optimal TSP tour. Furthermore, the algorithm guarantees that the resulted TSP tour is not longer than the double path of the MST, and the quality of resulted tour T(n) is T(n) <= 2OPT(n). Thus the MST algorithm also gives us an upper bound of the TSP tour. Meanwhile, The complexity of the MST algorithm is dominated by the complexity of the MST generation, which is $O(n^2)$. [Lawler, Lenstra, Kan and Shmoys 1985]

Based on the MST algorithm, another better heuristic is created: the Christofrides algorithm. [Aarts and Lenstra 1997] This algorithm requires a MST first for getting the set of degreed vertices. Then it computes a minimum length matching M on the vertices of odd degree in the MST, and connects M with the MST to get a new connected graph G'. G' is an Euler graph that includes an Euler tour. Within the Euler tour, a TSP tour can be easily found by traversing the Euler tour while using some shortcuts to replace some edges so as to avoid visiting a vertex more than once. With this algorithm, constructing the MST needs $O(n^2)$ and constructing a TSP tour needs $O(n^3)$. But the quality of the resulted tour T(n) is improved as T(n) <= 3/2OPT(n).

There are also some other methods to generate the TSP tour based on the MST. For example, there is a parallel algorithm used in those computers that have more than one processor. In this method, the technique for getting the TSP tour is similar with the techniques in the double MST algorithm. But its time cost is decreased to $O(n \log_2 n)$. [Dale and Walker 1996]

Beyond the MST algorithm for the TSP tour, there are many other heuristic algorithms for the TSP. Most of these heuristics can improve more the quality of the TSP tour than the MST algorithm does. These Heuristics can be roughly divided into two strategies. One is called the tour construction; the other is called the local search optimization. [Lawler, Lenstra, Kan and Shmoys 1985] [Aarts and Lenstra 1997]

The tour construction strategy is focused on generating a TSP tour gradually. Normally, the heuristics under this strategy start their computing at an arbitrary vertex on graph G. The heuristics will then select an optimal vertex one by one and connect it to the prior vertex or insert it into the prior existing sub-tour. The well-known algorithms in this group are: the nearest-neighbor algorithm, the insertion algorithm, the greedy algorithm and the Clarke-Wright algorithm. [Gu 1998] In fact, the MST algorithms for the TSP tour are also under this group. There are also other algorithms adopting a different strategy to do the tour construction. J. L. Bentley has suggested a bitonic tours method to solve the euclidean TSP, that is, constructing a TSP tour from the leftmost vertex on the euclidean plane and going strictly left to the rightmost vertex, and then going strictly right to left back to the starting vertex. [Cormen, Leiserson and Rivest 1995] P. Yager and J. P. Chandler presented the strip algorithm. The algorithm divides all points on the graph G into equal-width strips, then starts from the leftmost strip and visits the points in the strip with the "Boustrophedonic (as the ox plows) algorithm", and proceeds through the strips one by one until all points are connected in a cyclic tour. [Gu 1998]

The local search optimization strategy is focused on improving a starting tour repeatedly. All algorithms of this strategy are based on the dynamic programming discipline. They start generating a TSP tour from an arbitrary Hamiltonian tour no matter how it was constructed. From the start tour, local search algorithms do some improvement to generate a new and improved tour. This technique is called local optimization. Then the improved tour becomes a new start tour for next iteration of the tour improvement. Under this group, there are the simple local search algorithm, the simulated annealing algorithm, the Tabu search algorithm, the genetic algorithm and the neural network algorithm. [Johnson and McGeoch 1995]

The simple local search algorithm is a well-known algorithm. It uses the edge-exchange method to improve the start tour until no better result can be received. This algorithm takes time under $O(n^k)$, k is the number of the edges that are selected to be broken and exchanged. The simulated annealing algorithm and the Tabu search algorithm adopt the technique called "uphill moving" that accepts some worse local search results as the base of further searching. The uphill-move technique is expected to get the better solutions with a higher probability. The complexity of the Tabu search seems as $\Omega(n^3)$, and the running time of the simulated annealing is at least $\Theta(n^2)$. The genetic algorithm is generally a globe optimal algorithm. The TSP application of the genetic algorithm was at the first time presented by Brady in 1985. [Johnson and McGeoch 1995] It uses recombination techniques and the fitness measure to generate the better generation of some "population" until some limitation of the reproduction is reached. The last selected population is seemed approximately optimal. The neural network algorithm can be divided into two classes. The first organizes neurons based on some integer algorithm. The second views the neurons as the points in space and uses some geometric algorithm to approach a near-optimal TSP tour. The time consumption of the neural network algorithm ranges from $\Theta(n^4)$ to $\Theta(n^3)$. [Johnson and McGeoch 1995] [Noschang 1997] [Lawler, Lenstra, Kan and Shmoys 1985]

Some Conclusions Regarding Previous Work

In order to solve the TSP, many algorithms and heuristics have been created, but there are still possibilities to improve the algorithms' computing quality when they are applied to solve the TSP. Since there have been respectable mature algorithms invented for the TSP in the last several decades, many new algorithms or heuristics are currently created by the modification of some existing algorithms to improve the resulted quality of the original algorithms. Meanwhile, the combining of different algorithms is also a significant trend in the TSP field to create new algorithms. [Lawler, Lenstra, Kan and Shmoys 1985] [Martin and Otto 1994]

However, as far as the MST algorithm goes, there is hardly any effort to modify the MST algorithm so as to improve the computing quality of the MST algorithm for the TSP tour. Considering the MST algorithm's theoretical achievement – it provides us the approved upper bound and the lower bound, it is worth trying to find some fast, easy and efficient way to modify the algorithm, and getting some better computing quality in practice.

CHAPTER III

RESEARCH STATEMENT

This research is focused on a complete and Euclidean graph G. G is a set of vertices V and symmetric edges E. It satisfies the criterion of triangular inequality. Through research, two algorithms are innovated. The purpose of the designed algorithms is to modify the MST algorithm for a better approximate resolution of the TSP tour.

The basic idea of innovation of the new algorithms is to modify the MST algorithm by combining the MST algorithm with some heuristic methods. Here, the research is focused on the edge-exchange method and insertion method. The idea is to adopt the techniques of the two methods to modify the second step of the MST algorithm, so as to convert a MST to a TSP tour more efficiently. For implementing the new algorithms, we also need to adopt the techniques of searching, sorting and inserting, as well as to rely on the knowledge of greedy strategy and dynamic programming strategy.

Based on the above ideas, several tasks are formulated and listed below.

1. Design of the two new algorithms. The first adopts 4-opt technique to convert a MST to an approximately optimal TSP tour. The second adopts insertion technique to convert MST to TSP tour. The edge-exchange technique (k-opt) and the insertion technique are relatively mature and efficient techniques invented to solve the combinatorial problems including the TSP. Originally, both techniques start their

computation on an arbitrary or random base, and they use greedy iteration until there is no shorter tour to be found. They can get much better optimal results than the MST algorithm in practice, but they take much more computing time than the MST algorithm, especially in large-scale cases. Based on a random distribution of all vertices, both algorithms must do many trial-and-error computing to find the shortest connection among vertices. For example, if the edge-exchange algorithm wants to check all k-pairs combination of all random trial tours, it will take $O(N^k)$ time to complement the computation of the TSP tour, which is a long period of time.

There are some variants of the edge-exchange algorithm that start computation under some preconditions. For example, there are some empirical implementations of the algorithms that start edge-exchange at some pre-selected tours such as the nearest neighbor tour, the greedy tour and the Clarke-Wright tour. [Lawler, Lenstra, Kan and Shmoys 1985] [Aarts and Lenstra 1997] However, all these empirical computations do not select a MST tour as their start tour. As a new alternative of the edge-exchange technique, our first new algorithm is designed to select a MST tour as the start tour before using the edge-exchange method to generate a better TSP tour.

Meanwhile, the insertion algorithm uses dynamic programming philosophy to generate an optimal TSP tour. There are also several variations of this algorithm. All of them start at a trial-smallest optimal tour visiting the first three arbitrary vertices that are distributed randomly. Then the algorithm will insert new vertex one after another into the tour, and use the branch-and-bound algorithm continuously to get the new optimized tour until all vertices are included. This technique needs a time cost of $O(n^2)$. [Dakin 1997] The difference between this technique and our second new algorithm is that our algorithm starts insertion from a minimum spanning tree that is a set of the ordered vertices generated by the MST algorithm. The purpose of this design is to keep the advantage of the MST, and to expect a better result than the one from the original MST algorithm.

2. Implementing the coded applications of the new algorithms and perform experiments with the new algorithms to see if the results satisfy the expectations. The expected results should shorten the tour length, while not increasing time cost. It is assumed that, in the worst case, these new algorithms will return the same results as the original MST algorithm. Generally, the innovated algorithms will get better results.
3. For checking the results of the new algorithms, we will compare them with the results of the MST algorithm with preordered walk method, a random 3-opt algorithm and a random insertion algorithm. First of all, experiments of these algorithms will be done with various pseudo-random data inputs. Then the statistic data of the output will be collected and some comparisons will be analyzed based on the statistic data. Our conclusion will be based on these empirical results.
4. Implementing a Visual Basic GUI of the experiment to have the computations and the results visible and more understandable.

The significance of the research will be discussed at the end of this section. Simply speaking, the modification of the MST algorithm for the TSP tour will let us apply the MST algorithm better, and take advantage of the MST algorithm more efficiently. In addition, since the MST algorithm for the TSP tour is a typical example in many textbooks for demonstrating the TSP, our research can be applied as a teaching application to help students understand and think about algorithms deeply and actively.

The reasons why the new algorithms focused on the MST algorithm are:

1) There are some algorithms to generate MST relatively quickly (the least is O(n)) [Oliver 1993], so it will make total complexity more efficient.

2) We can evaluate the experimental results of the new algorithms easily with the MST. Since we know the total length of the minimum spanning tree (MST) is surely less than the length of its optimal tour, we can use the length of the MST as the lower bound of the real optimal tour, and apply the length of the MST to compare with, and evaluate, the results of the new algorithms.

CHAPTER IV

IMPLEMENTATION DESIGN

Basic Assumptions

In this research, the basic experimental computing of the designed algorithms will be carried out by a pseudo-random method. It means that the implemented programs of the algorithms will create all vertices (their coordinates) via a pseudo-random method. All the coordinates of the vertices will be valued in the range of (0, 1). In addition, the programs will also accept the input data from a given file, or from manual drawing.

The experimental output of the research will be available for basic statistic analysis. The main point is to find the average trend or the general quality of the results.

In this research, we will use C to develop the programs of all of the algorithms. Since most of TSP algorithms were implemented with C, C++ and Java, using C makes it available to compare our research results with that from other algorithms.

Otherwise, we will use Visual Basic to construct a visual interface to run all of the algorithms and present the results visually. Selecting Visual Basic to implement GUI is based on personal interest, and also on the fact that no one in OSU has used Visual Basic for this purpose before.

All algorithms will be run in a single processor computer with Windows NT platform, and with Visual Basic application support (VB5 or newer version).

New Algorithm 1

From Figure 1 that represents a TSP tour generated by the MST algorithm, we can find that some connections of vertices are apparently not optimal. These non-optimal connections are generated by the preordered walk method of the MST algorithm. If we can rearrange these non-optimal connections, we could generate a shorter tour to improve

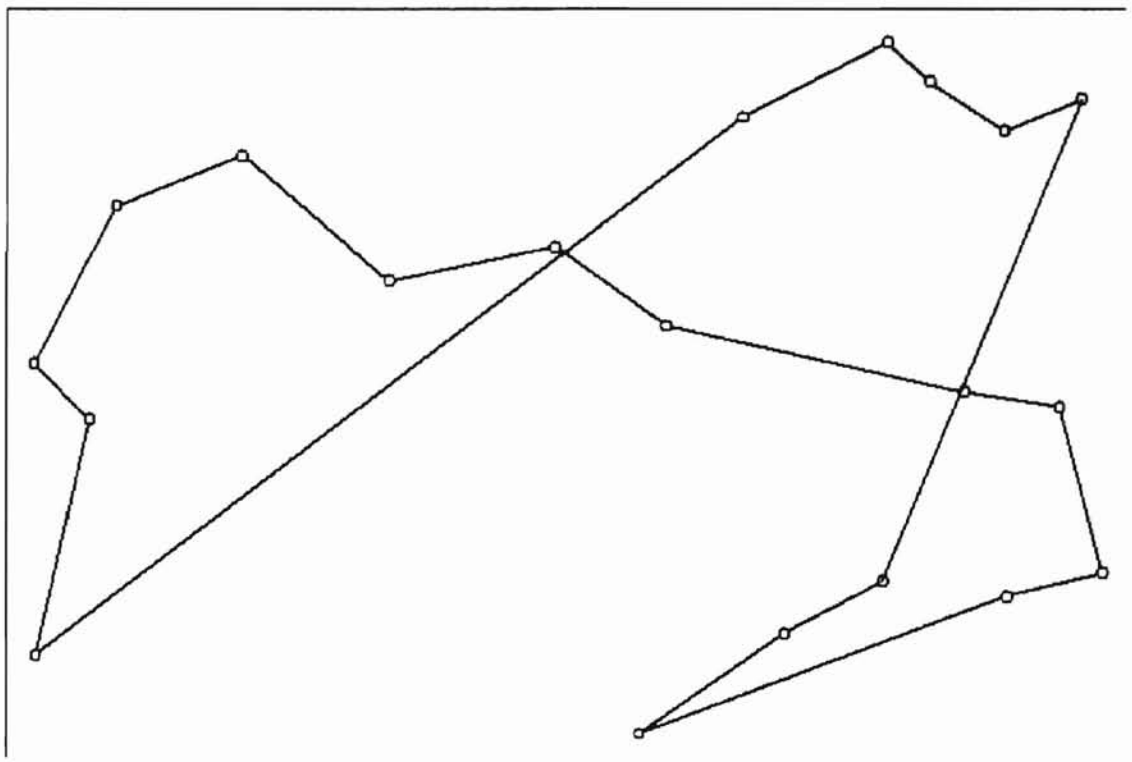


Figure 1: A TSP Tour Based on the MST Algorithm

the quality of the result of the MST algorithm. Based on this idea, the algorithm New1 is designed.

The design of New1 is based on the edge-exchange technique. First, we just generate a MST by using the preordered walk method to get a normal TSP tour. In this TSP tour, we assumed that some non-optimal paths must be existed and they must be the ones of the longest paths in the tour. Next, we use the edge-exchange method to break several of the first few longest paths in that starting tour, and try to rearrange and connect those broken points. If we can use this technique to find a new and shorter TSP tour, then we can replace the former longer tour with this new tour. This algorithm will be repeated based on branch-and-bound algorithm until there is no further shorter tour could be found. The complexity of this method should be less than O(k), k is the number of the iteration. The worst case of the New1 will be same as the original MST algorithm since the upper bound of the New1 tour is the tour resulted from the original MST algorithm.

Steps of Algorithm New1

1. Construct a MST S by the Prim's algorithm and the heap sort
2. Construct a Hamiltonian tour T of S using the preorder-walking method
3. Count the length L of T as the value of the upper bound
4. Find the four longest edges of T
5. Break the four longest edges to get four sub paths
6. Use 4-opt method to generate a new tour T' shorter than the upper bound
7. If we can find T', let T <- T', up-bound = the length L' of T'
8. Repeat 3 - 7
9. Return T as the result when we cannot find T'

New Algorithm 2

The design of the algorithm New2 is based on the insertion technique. The difference between the algorithm New2 and the common insertion algorithms is obvious. The latter normally use randomly selected vertices on graph G to do insertion. But our algorithm New2 is based on a MST. The MST provides an ordered set of all vertices to start the insertion operation and finally to complement a TSP tour. The complexity of the insertion of the algorithm New2 should be $O(n^2)$ and its result will be better than the one of the MST algorithm.

Steps of Algorithm New2

1. Construct a MST S by the Prim's algorithm and the heap sort
2. Select the first three nodes in S to construct the beginning sub tour T
3. Insert the next node in S into T, based on the technique that will find the most proper position of it in T so as to generate the new shortest tour T'
4. Let T <- T'
5. Repeat step 3 – 4 until all nodes in S is inserted in T
6. Return T as the result

CHAPTER V

EXPERIMENT RESULTS

For comparison of the results of the new algorithms with the original MST algorithm, we ran all of the algorithms with the pseudo-random cases in a set of 40, 80, 160, 320, 640 and 1280 cases, and iterated every case for 100 times. Also, for evaluating the quality of the results of the new algorithms, we ran two heuristics, the random 3-opt algorithm and the random insertion algorithm under the above conditions. However in the 640 and 1280 cases, 3-opt algorithm was iterated 5 times since the running time was too long. The experimental result is represented as the Figure 2.

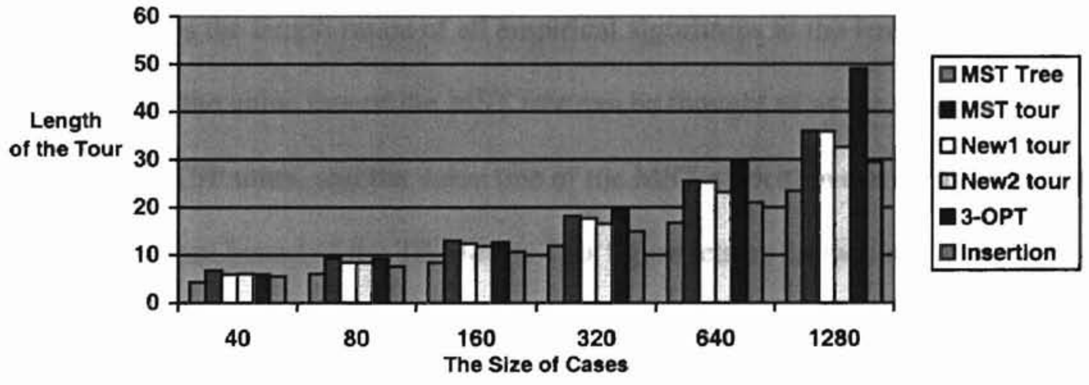


Figure 2: The Average Length of the TSP tours Generated by Different Algorithms

From Figure 2, it can be observed that our algorithms New1 and New2 can get shorter tours in all designed cases than the tours generated by MST algorithm. There are, however, different improvement trends between New1 and New2.

For algorithm New1, its improvement is depended on the size of the case. In small size cases (number less than 40), it can improve the tour quality by about 12%. In some individual cases the improvement can be as high as 25%. But along with the number of the case increasing, its improvement will decrease and when the number of the case is more than one thousand, there was no significant improvement.

For algorithm New2, its achievement is not as good as that of the algorithm New1 in small size cases. But in the mid-size cases (number around 320), it can improve the tour quality by about 15% and in some individual cases the improvement can be as high as 35%. On the whole, it demonstrates a good and stable improvement (around 9%) in all large-size cases.

Figure 3 and Figure 4 (see below) show the achievement of the new algorithms. Figure 3 shows the length ratios of all empirical algorithms to the length of the MST tree. In this figure, the value line of the MST tree can be thought of as the theoretical lower bound of the TSP tours, and the value line of the MST algorithm can be thought of as the theoretical upper bound of the TSP tours. This figure lets us know how far the new algorithms are from the lower bound and from the upper bound.

Figure 4 shows the improvement ratios of the new algorithms and other two selected heuristics to the result of the MST algorithm. This figure lets us know how much improvement the new algorithms have made to the MST algorithm.

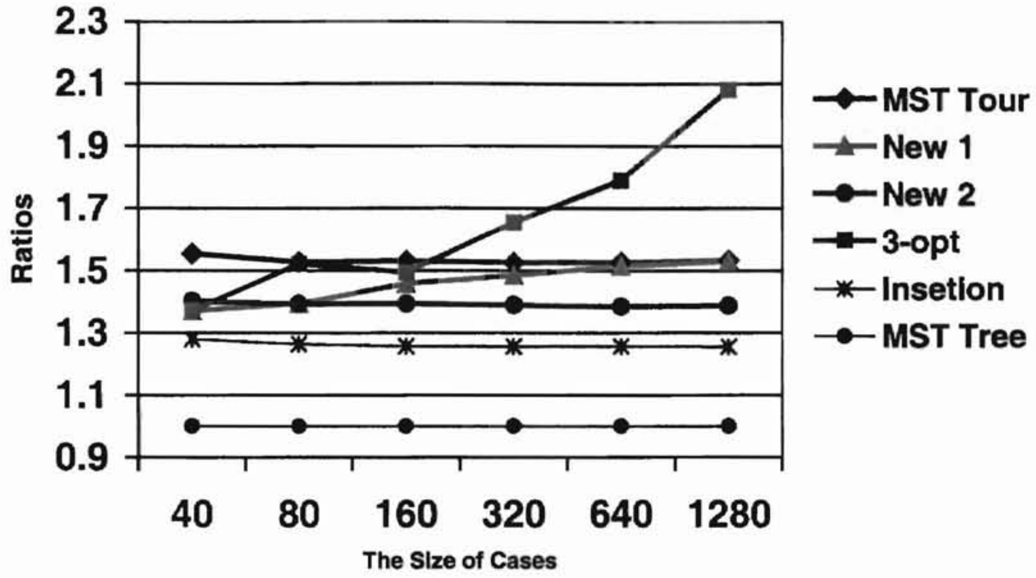


Figure 3: The Length Ratio of the Empirical Algorithms to the MST Tree

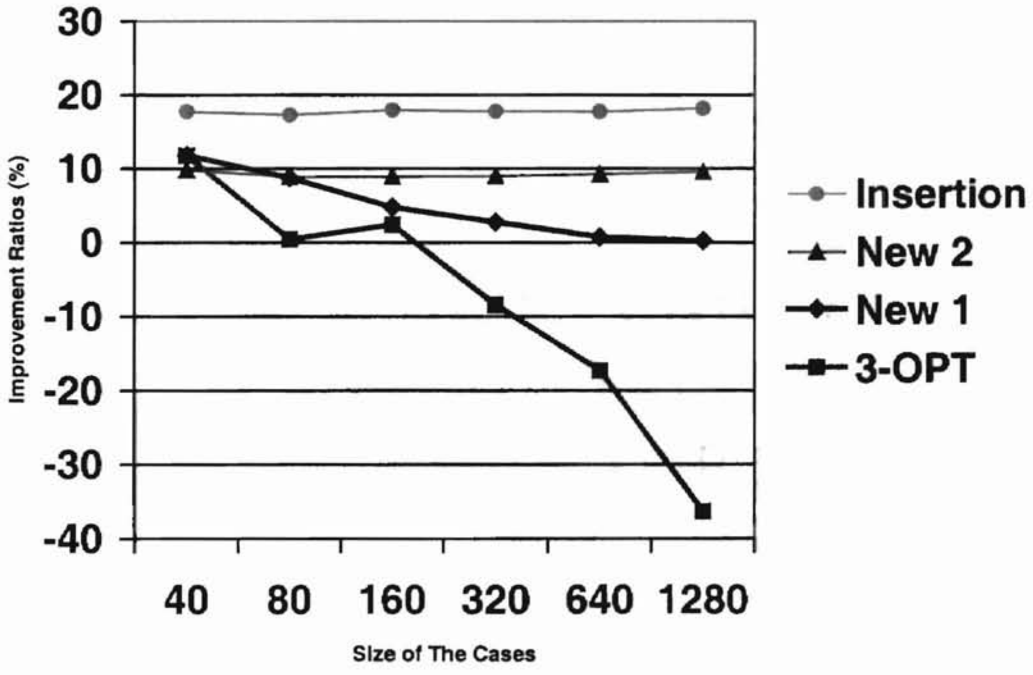


Figure 4: The Improvement Ratios of the New Algorithms to the MST Algorithm

Meanwhile, we compare the new algorithms with the random 3-opt algorithm and the random insertion algorithm. These two heuristic methods are implemented based on Dr. Chandler's project assignment [Chandler 1998] and Robert Dakin's article. [Dakin 1997]

Figure 5 and Figure 6 show the results of the comparison of our new algorithms to the two selected heuristics.

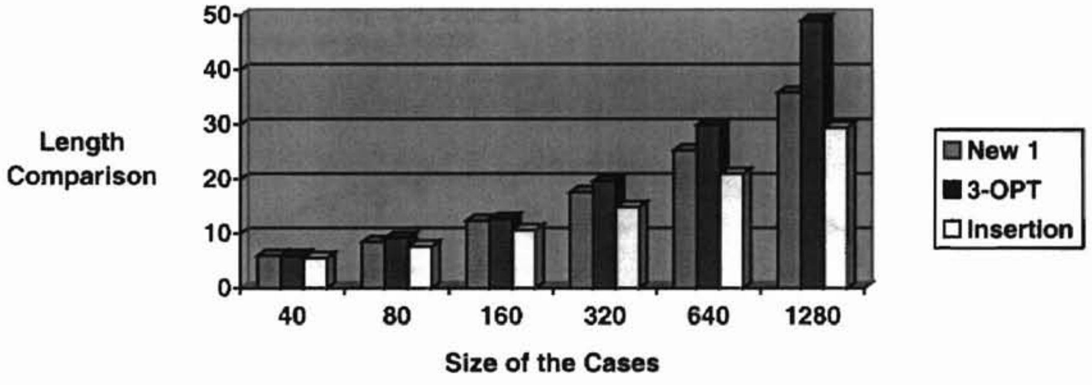


Figure 5: Comparison of the New Algorithm New1 to the Selected Heuristics

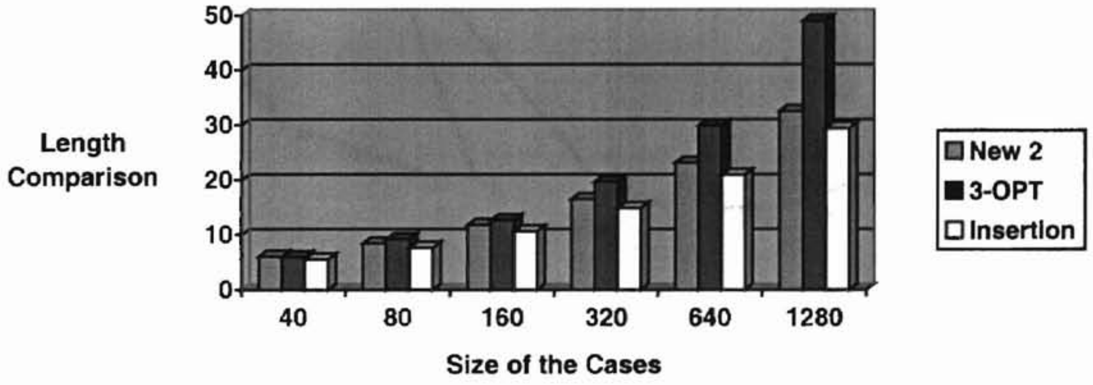
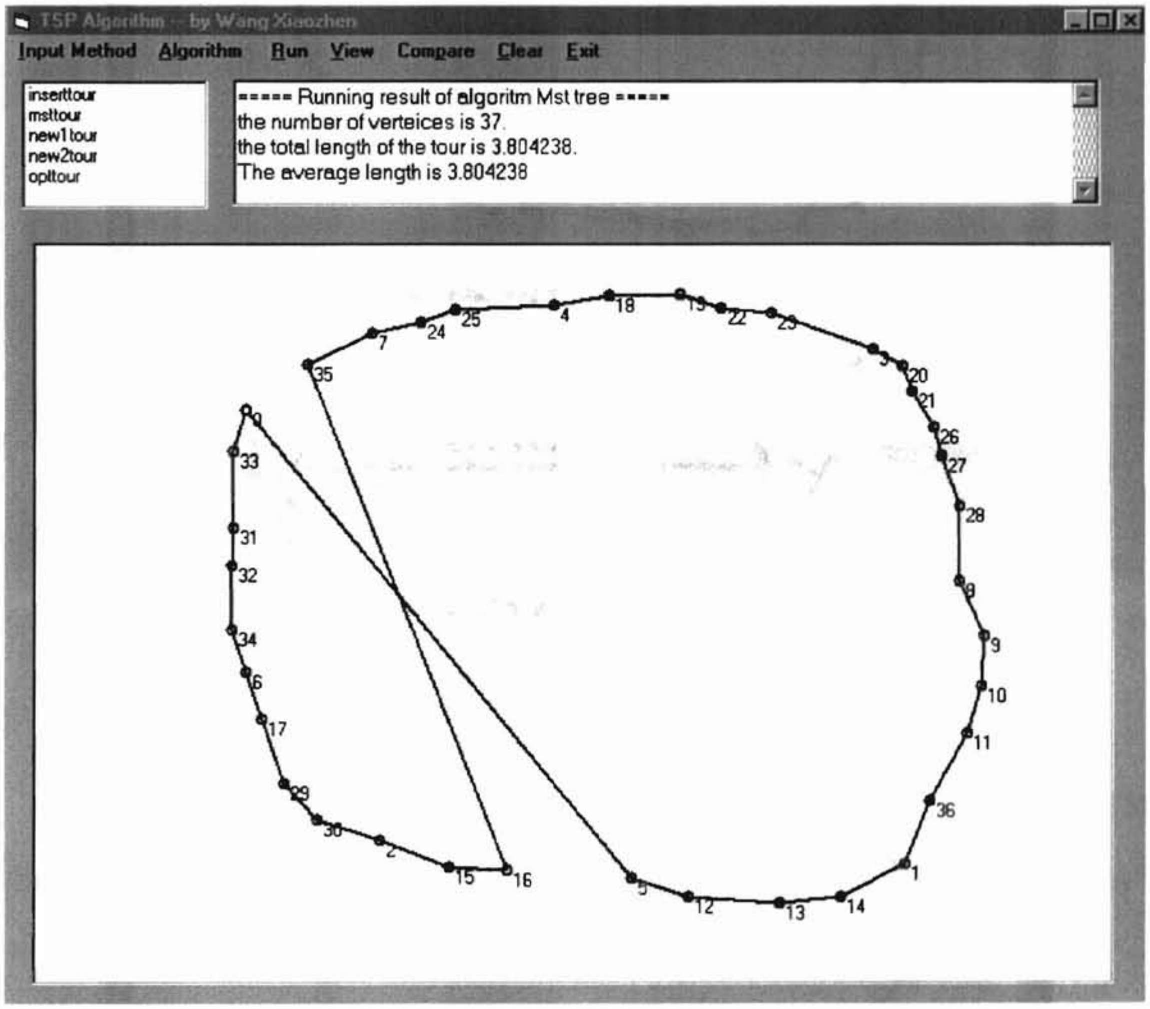


Figure 6: Comparison of the New Algorithm New2 to the Selected Heuristics

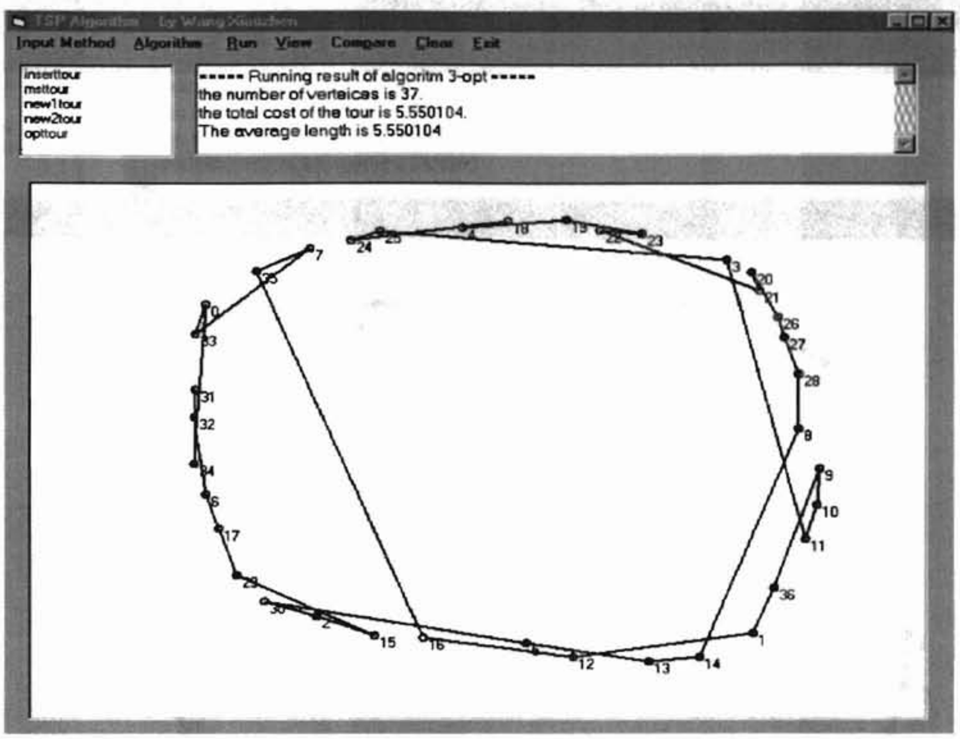
From these figures we know that our new algorithms are better than the random 3-opt algorithm but the random insertion algorithm is better than both the new algorithms.

However, in some arbitrary cases, our new algorithms can generate much better results than both heuristics. Figure 7 to Figure 10 demonstrated that our new algorithms generated the real optimal tour and the random 3-opt algorithm did not.



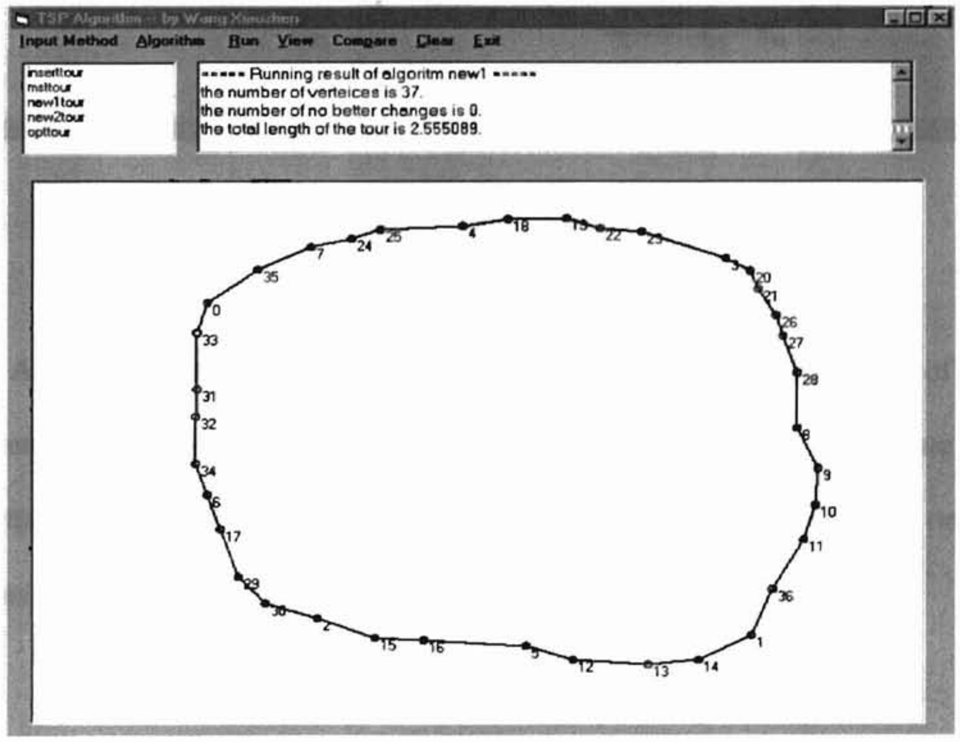
```
===== Running result of algoritm Mst tree =====
the number of verteices is 37.
the total length of the tour is 3.804238.
The average length is 3.804238
```

Figure 7: A TSP Tour Generated by the MST Algorithm (Preorder Walking)



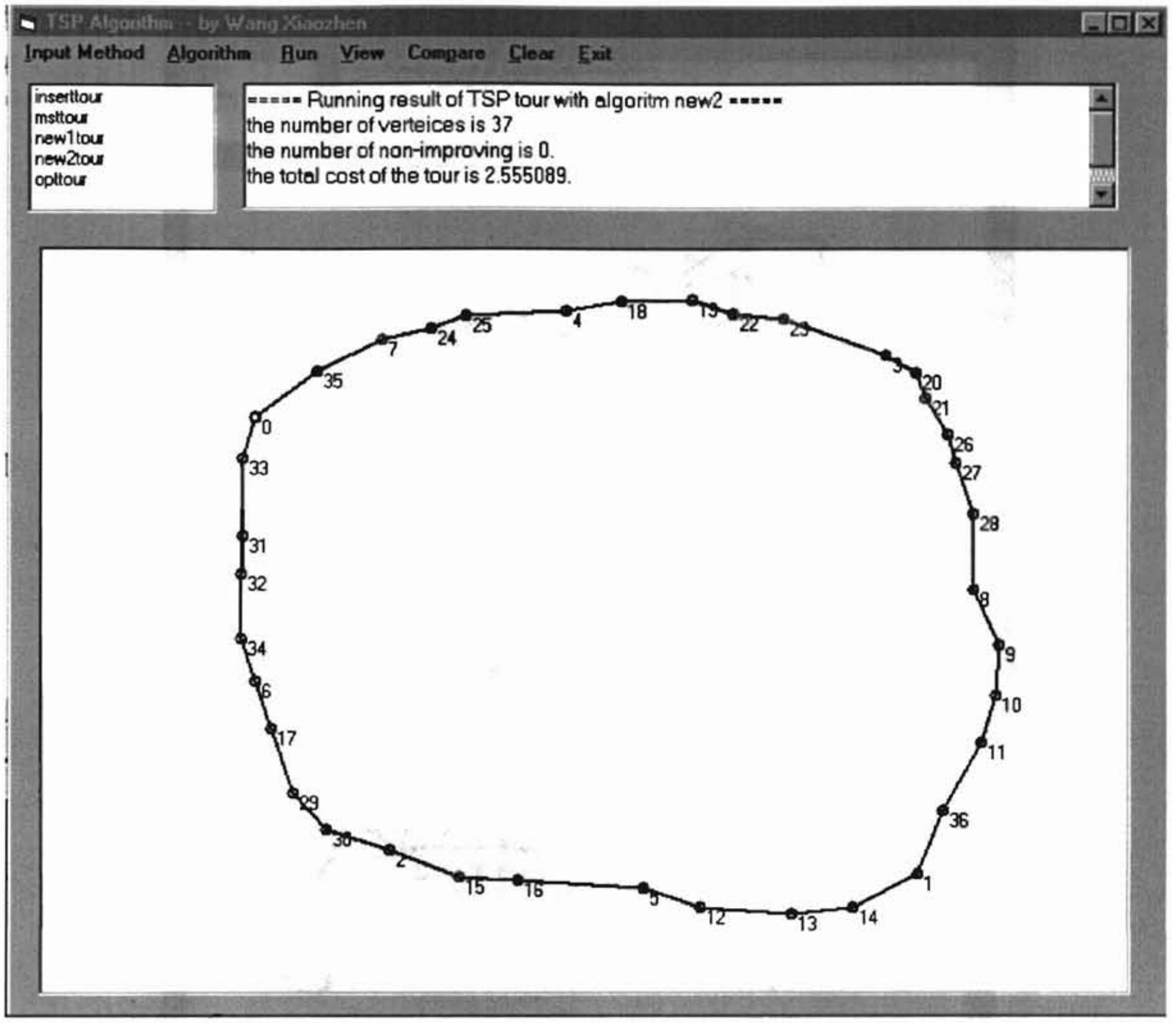
```
----- Running result of algoritm 3-opt -----
the number of verteices is 37.
the total cost of the tour is 5.550104.
The average length is 5.550104
```

Figure 8: A TSP tour Generated by the 3-opt Algorithm



```
----- Running result of algoritm new1 -----
the number of verteices is 37.
the number of no better changes is 0.
the total length of the tour is 2.555089.
```

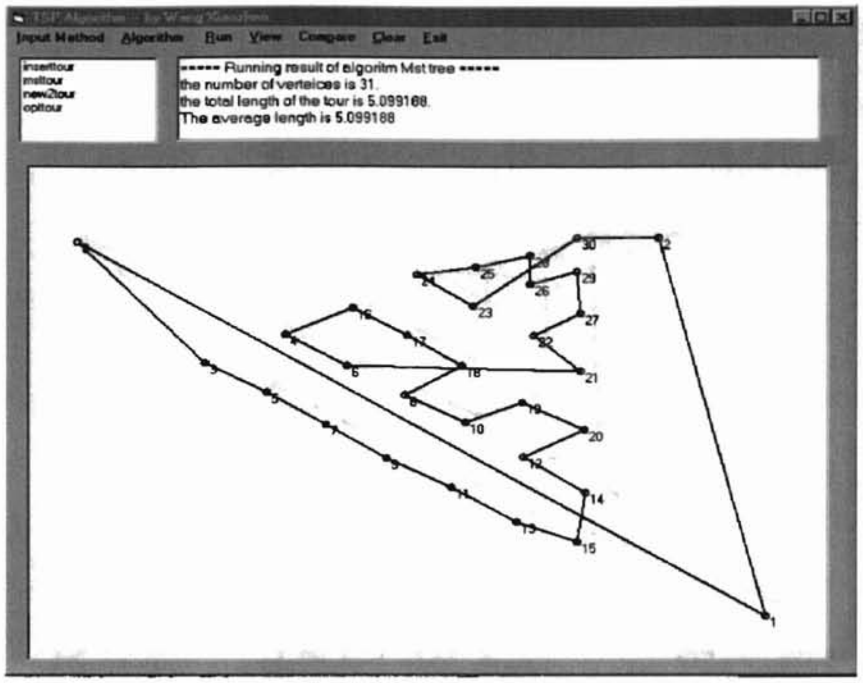
Figure 9: A TSP tour Generated by the New1 Algorithm



```
===== Running result of TSP tour with algoritm new2 =====
the number of verteices is 37
the number of non-improving is 0.
the total cost of the tour is 2.555089.
```

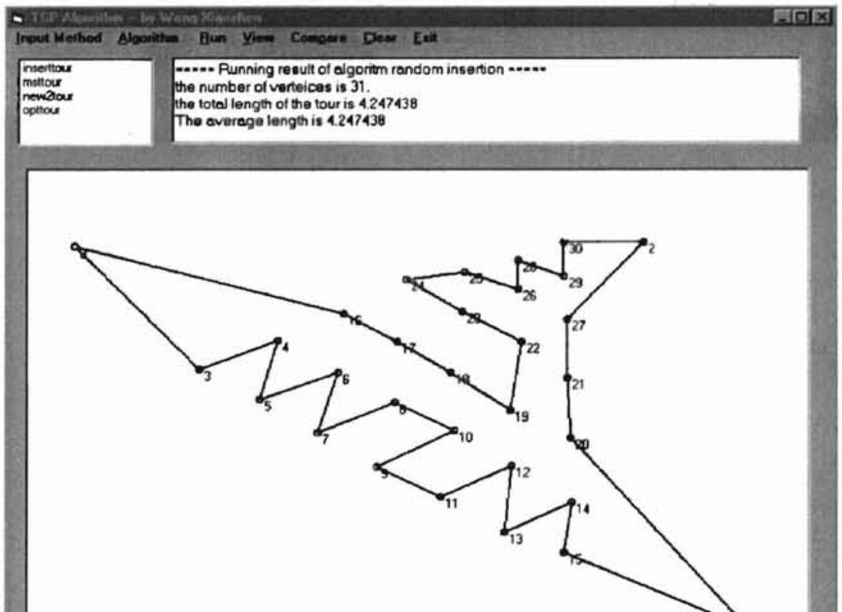
Figure 10: A TSP tour Generated by the New2 Algorithm

Although the random insertion algorithm generally gets the best result of the TSP tour in our experiment, in some arbitrary cases the algorithm New2 can generate better results than the random insertion algorithm does. Figure 11 to Figure 14 show one of these cases.



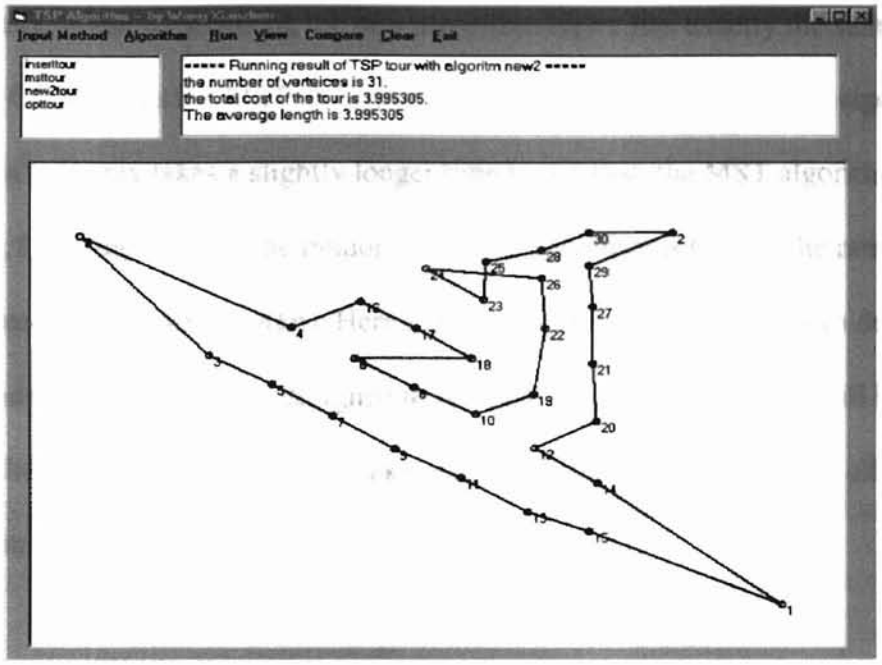
```
----- Running result of algoritm Mst tree -----
the number of verteices is 31.
the total length of the tour is 5.099188.
The average length is 5.099188
```

Figure 11: A TSP Tour generated by the MST Algorithm (Preorder Walking)



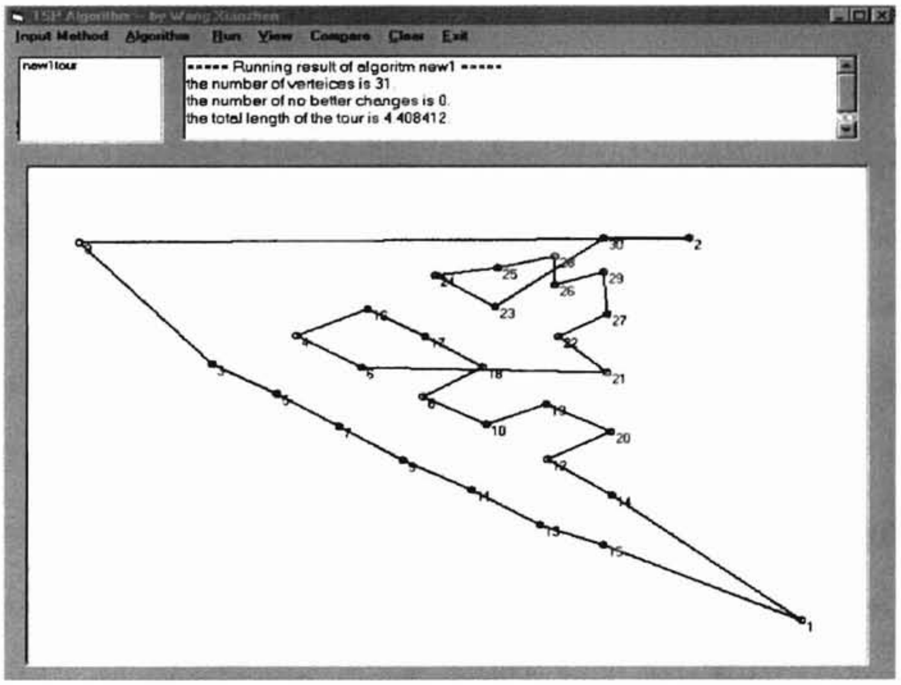
```
----- Running result of algoritm random insertion -----
the number of verteices is 31.
the total length of the tour is 4.247438
The average length is 4.247438
```

Figure 12: A TSP Tour Generated by the Random Insertion Algorithm



```
----- Running result of TSP tour with algoritm new2 -----
the number of verteices is 31.
the total cost of the tour is 3.995305.
The average length is 3.995305
```

Figure 13: The TSP Tour Generated by the New Algorithm New2



```
----- Running result of algoritm new1 -----
the number of verteices is 31.
the number of no better changes is 0.
the total length of the tour is 4.408412.
```

Figure 14: A TSP Tour Generated by the New Algorithm New1

Finally, in our experiment, the new algorithm New1 has exactly the same complexity O(nlg(n)) as the MST algorithm, and the complexity of the new algorithm New2 is $O(n^2)$. It only takes a slightly longer time to run than the MST algorithm. Meanwhile, the complexity of the random insertion algorithm is $O(n^2)$. The random 3-opt algorithm has the complexity O(kn). Here k is the number of the iterations to do the edge breaking and exchanging. If k is designed to be equal to n, the complexity will be $O(n^2)$. Otherwise the random 3-opt algorithm takes much more memory space than all other algorithms in our experiment.

CHAPTER VI

SUMMARY AND CONCLUSIONS

In this research, we implemented both the new algorithms and performed empirical running of them based on the pseudo-random input data. We also implemented a VB GUI to demonstrate the results of the experiment.

Through the whole experiment of the new algorithms, we found that the new algorithms have truly improved the quality of the MST algorithm. Explicitly speaking, the new algorithm New1 can apparently improve the quality of the original MST algorithm in small-size cases; and the new algorithm New2 can apparently improve the quality of the original MST algorithm in mid- and large-size cases. Although the new algorithms can achieve better results than the original MST algorithm in practice, they still cannot change the upper bound of the MST algorithm since they are limited by the property of the MST tree.

Otherwise, in our experiment, we found that the random-insertion algorithm has ,the best practical result although its upper bound cannot be decided explicitly, and our new algorithms are generally better than the random 3-opt algorithm.

Therefore, the results of the new algorithms satisfied the expectations of the design. But the improvement to the MST algorithm is not good as we have expected, especially as far as the new algorithm New1 goes. Our experiment also demonstrated the fact that the modification of the MST algorithm is always limited by the way of the combination of vertices under the MST tree, so it is difficult to break the upper bound of the MST algorithm.

REFERENCES

[Aarts and Lenstra 1997] Emile Aarts and Jan Karel Lenstra, *Local Search in Combinatorial Optimization*, John Wiley & Sons, Chichester, England, 1997

[Anderson 1996] E. J. Anderson, Mechanisms for Local Search, *European Journal of Operational Research*, Vol. 88 (1996) pp. 139 - 151

[Atallah 1999] Mikhail J. Atallah, *Algorithms and Theory of Computation Handbook*, CRC Press, 1999

[Cannon 1984] Kenneth Andrew Cannon, Least Squared Approach To The Traveling Salesman Problem, Master's Report, Business Administration, Oklahoma State University, 1984

[Chandler 1998] John P. Chandler, COMSC 5413 Program 3: Traveling Salesman problem, Computer Science, Oklahoma State University, 1998

[Chatterjee, Carrera and Lynch 1996] Sangit Chatterjee, Cecilia Carrera, and Lucy A. Lynch, Genetic Algorithms and traveling Salesman Problems, *European Journal of Operational Research*, Vol. 93 (1996) pp. 490 – 510

[Cormen, Leiserson and Rivest 1995] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction To Algorithms*, The MIT Press and McGraw-Hill Book Company, New York, 1995

[Dakin 1997] Robert Dakin, The Travelling Salesman Problem, 1996-1997 http://www.pccug.org.au/~dakin/tsp.html

[Dakin 1997] Robert Dakin, A Simple Insertion Algorithm, 1996-1997
http://www.pccug.org.au/~dakin/tspbb.html

[Dale and Walker 1996] Nell Dale and Henry M. Walker, *Abstract Data Types – Specifications, Implementations, and Applications*, D. C. Heath and Company, Massachusetts, 1996

[Fredman, Johnson, McGeoch and Ostheimer] M. L. Fredman, D. S. Johnson, L. A. McGeoch and G. Ostheimer, Data Structures for Traveling Salesmen,
http://www.ing.unlp.edu.ar/cetad/mos/TSPBIB_home.html

[Garey and Johnson 1979] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness*, W. H. Freeman, San Francisco, 1979

[Ghalib 1979] Hani M. Ghalib, The Traveling Salesman Problem: Literature Survey and Comparison of Two Computational Methods, Master's Report, Computer Science, Oklahoma State University, 1979

[Gu 1998] Yan Gu, Animation Of Approximation Algorithms For Traveling Salesman Problem In Java, M. S. Thesis, Computer Science, Oklahoma State University, 1998

[Johnson and McGeoch 1995] David S. Johnson and Lyle A. McGeoch, The Traveling Salesman Problem: A Case Study in Local Optimization, 1995
http://www.ing.unlp.edu.ar/cetad/mos/TSPBIB_home.html

[Jünger, Reinelt and Thienel 1994] Michael Jünger, Gerhard Reinelt, and Stefan Thienel, Provably Good Solutions For the Traveling Salesman Problem, revised version of June 1993, 94_31ps.ps, Preprint IWR, Heidelberg, 1994
http://www.ing.unlp.edu.ar/cetad/mos/TSPBIB_home.html

[Kureichick, Miagkikh and Topchy] Victor M. Kureichick, Victor V. Miagkikh and

Alexander P. Topchy, Genetic Algorithm for Solution of the Traveling Salesman Problem with New Features Against Premature Convergence, http://www.ing.unlp.edu.ar/cetad/mos/TSPBIB_home.html

[Lawler, Lenstra, Kan and Shmoys 1985] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan and D. B. Shmoys, *The Traveling Salesman Problem – A Guided Tour of Combinatorial Optimization*, John Wiley & Sons, 1985

[Lin 1965] Shen Lin, Computer Solutions of the Traveling Salesman Problem, *Bell System Technical Journal*, Vol. 44 (1965), pp. 2245 – 2269

[Manber 1989] Udi Manber, *Introduction To Algorithms – A Creative Approach*, Addison-Wesley Publishing Company, 1989

[Martin and Otto 1994] Olivier C. Martin and Steve W. Otto, Combining Simulated Annealing with Local Search Heuristics, 1994 http://www.ing.unlp.edu.ar/cetad/mos/TSPBIB_home.html

[Miller, Tucker and Zemlin 1960] C. E. Miller, A. W. Tucker and R. A. Zemlin, Integer Programming and the Traveling Salesman Problem, *Journal, Association of Computing Machinery* Vol. 7 (1960), pp. 326 - 329

[Moscato 1998] Pablo Moscato, TSPBIB Home Page, 1998 http://www.ing.unlp.edu.ar/cetad/mos/TSPBIB_home.html

[Noschang 1997] Mark H. Noschang, The Traveling Salesman Problem – a review of theory and current research, 1997 http://www.ececs.uc.edu/~mnoschan/sale.html

[Oliver 1993] Ian Oliver, *Programming Classics – Implementing the World's Best Algorithms*, Prentice Hall, New York, 1993

[Papadimitrious and Steiglitz 1977] C. Papadimitrious and K. Steiglitz, On the Complexity of Local Search for the Traveling Salesman Problem, *SIAM Journal on Computing*, Vol. 6 (1977), No. 1, pp. 76 – 83

[Smith 1987] Harry F. Smith, *Data Structures – Form and Function*, Harcourt Brace Jovanovich, Publisher, 1987

[Voudouris and Tsang 1995] Chris Voudouris and Edward Tsang, Guided Local Search, *Technical Report CSM-247*, 1995

http://www.ing.unlp.edu.ar/cetad/mos/TSPBIB_home.html

[Willard 1970] Edward Payson Willard, An Algorithm for Optimal Ship Routing For Seismic Data Collection, Ph.D. Thesis, Industrial Engineering and Management, Oklahoma State University, 1970

[Yagiura and Ibaraki 1996] Mutsunori Yagiura and Toshihide Ibaraki, The Use of Dynamic Programming in Genetic Algorithms for Permutation Problems, *European Journal of Operational Research*, Vol.92 (1996) pp. 387 – 401

[Zweig 1995] Geoffrey Zweig, An Effective Tour Construction and Improvement Procedure for the Traveling Salesman Problem, *Operations Research*, Vol. 43 (1995) pp. 1049 – 1057

APPENDIX

Table1: The Improvement Results to the MST Algorithm

Num. of Vertices	40	80	160	320	640	1280
MST	6.746908	9.269029	12.996129	18.164780	25.530018	35.975517
New1	5.950680	8.457107	12.374555	17.659966	25.336664	35.894157
Improving %	11.80	8.76	4.78	2.78	0.76	0.23
New 2	6.084479	8.446555	11.834032	16.533146	23.169161	32.551655
Improving %	9.82	8.87	8.94	14.49	9.25	9.52
3-opt	5.953973	9.228397	12.685232	19.697220	29.944220	49.043861
Improving %	11.75	0.44	2.39	-8.44	-17.29	-36.33
Insertion	5.551500	7.667203	10.666630	14.942585	21.007906	29.444775
Improving %	17.72	17.28	17.92	17.74	17.71	18.15

Table2: The Ratio to the Lower-bound (MST Tree)

Num. of Vertices	40	80	160	320	640	1280
MST Tree	4.338862	6.064266	8.489203	11.903287	16.721451	23.461361
MST	6.746908	9.269029	12.996129	18.164780	25.530018	35.975517
Ratio	1.555	1.528	1.531	1.526	1.527	1.533
New 1	5.950680	8.457107	12.374555	17.659966	25.336664	35.894157
Ratio	1.371	1.395	1.458	1.484	1.515	1.530
New 2	6.084479	8.446555	11.834032	16.533146	23.169161	32.551655
Ratio	1.402	1.393	1.394	1.389	1.384	1.387
3-opt	5.953973	9.228397	12.685232	19.697220	29.944220	49.043861
Ratio	1.372	1.522	1.494	1.655	1.791	2.082
Insertion	5.551500	7.667203	10.666630	14.942585	21.007906	29.444775
Ratio	1.279	1.264	1.257	1.255	1.256	1.255

Table3: The Improvement of the New1 to the 3-opt and Insertion Algorithms

Num. of Vertices	40	80	160	320	640	1280
New 1	5.950680	8.457107	12.374555	17.659966	25.336664	35.894157
3-opt	5.953973	9.228397	12.685232	19.697220	29.944220	49.043861
Improving to 3-opt	0.06%	8.36%	2.45%	10.34%	15.39%	26.50%
Insertion	5.551500	7.667203	10.666630	14.942585	21.007906	29.444775
Improving to Insertion	-7.19%	-10.30%	-16.01%	-18.19%	-20.61%	-21.90%

Table4: The Improvement of the New2 to the 3-opt and Insertion Algorithms

Num. of Vertices	40	80	160	320	640	1280
New 2	6.084479	8.446555	11.834032	16.533146	23.169161	32.551655
3-opt	5.953973	9.228397	12.685232	19.697220	29.944220	49.043861
Improving to 3-opt	-2.19%	8.47%	6.71%	16.06%	22.63%	33.35%
Insertion	5.551500	7.667203	10.666630	14.942585	21.007906	29.444775
Improving to Insertion	-9.60%	-10.17%	-10.94%	-10.64%	-10.29%	-10.55%

Main Programming Codes of Implementation

```
//==========================================================================
```

General Functions

```
void heapfy(vtx *pa, int i, int size)
{
 int l, r, hsize, large;
 vtx temp;

 l = left(i);
 r = right(i);
 hsize = size;
 large = i;

 if(l<hsize)
 {
  if((pa+l)->cost<(pa+large)->cost)
    large = l;
 }

 if(r<hsize)
 {
  if((pa+r)->cost<(pa+large)->cost)
    large = r;
 }

 if(large!=i)
 {
  temp = pa[i];
  pa[i] = pa[large];
  pa[large] = temp;
  heapfy(pa, large, size);
 }

}
//==========================================================================

void heapsort(vtx *pa, int size)
{
  int i;
  vtx temp;

  for(i=(size/2-1); i>=0; i--)
    heapfy(pa, i, size); //get the shortest edge and put it in pa[0]
 temp = pa[0]; //change the position of the shortest edge to the end of pa
 for(i=0; i<size-1; i++)
   pa[i] = pa[i+1];
 pa[i] = temp; //put the shortest edge in pa[i]
}
//========================================================================

float urand()
{
 int Q, R;
 static int Seed = 56789;
 int M = 2147483647;
 int A = 16807;
 float URand;

 Q = M / A;
 R = M % A;

 do
 {
  Seed = A * (Seed % M);
  Seed = A * (Seed % Q) - R * (Seed / Q);
  if(Seed < 0)
   Seed = Seed + M;
  URand = (float)Seed / M;
 }
 while (!((URand >0.0) && (URand <1.0)));
 return URand;
}
//========================================================================
```

Program for generating a minimum spanning tree

```
int main(int argc, char * argv[])
{
 int j, k, n, repeat, nochange, flag, size;
 float sumcos, aversum, longest, shortest;
 char title1[80] = "========== Running Results of MST Tree ==========";
 char title2[80] = "========== Paths of the Resulted MST Tree ==========";

 char origcity[30] = "";
 char treetour[30] = "C:\\wxiaozh\\msttree";
 char treesum[30] = "C:\\wxiaozh\\treesum";
 char treedata[30] = "C:\\wxiaozh\\treedata";

 vtx *origi;
 vtx *que;
 vtx *mst;

 if (argc<3)
 {
  printf("Running command should be as following example: \n");
  printf("execution-file input-file number-of-vertex [number-of-iterate]\n");
  printf("  *.exe     inputfile    integer       [interger] \n");
  exit (0);
 }
```

```
j = atoi(argv[2]); //store the number of vertex
strcpy(origcity, argv[1]); //get the input file
n = 1; //initial the iteral number as 1
if (argc>3) n = atoi(argv[3]); //if fourth argument, get the number of iterate

repeat = n;
nochange = 0;
flag = 1;
longest = 0.0;
shortest = 999999.0;
aversum = 0.0;

origi = dynmem(j); //set memory for computing
que = dynmem(j);
mst = dynmem(j);

while(n>0) //this loop is for iterate times of executing
{
        if (n == 1) //set vertex array with input data file
                setvertices(origcity, origi, j);
        else //initial the array of vertices with internal random function
        setvertex(origi, j);

  setstart(que, origi, j); //select the start vertex
  initmst(mst, que, j); //set the first node and its value in MST tree
  size = j-1;
  k = 0;
                //generating a MST tree in the following while loop
  while(size>0)  //follows maybe not right! Check it.
  {
    setvertexcost(mst, que, k, size);
    heapsort(que, size);              //sort and find the next shortest node
    k += 1;
    size -= 1;
    mst[k] = que[size];      //put the next shortest node in mst
  } //the end of while(size>0)

  sumcos = treelength(mst, j); //count the total lengths of the MST tree
  aversum = aversum + sumcos;
  if(sumcos>longest) longest = sumcos;
  if(sumcos<shortest) shortest = sumcos;

  n--;

} //the end of while (n>0)   flag = -1;

nochange = -1;
aversum = aversum/repeat;
writetree(treetour, mst, j); //write an data file of all vertices coordinates
printdata(treesum, title1, sumcos, aversum, longest, shortest, repeat, nochange, j);

free(origi);
free(que);
free(mst);
```

```
} //main's end
//================================================================
```

Program for implementing the MST algorithm

```
void setmst(vtx *sourse, vtx *destinate, int size)
{
 int start, reset = -1;
 int i = 0, k = 0, count = 0;

 do
 {
  if(sourse[i].idx > 0) //the node is available
  {
        sourse[i].idx = 0;
    destinate[count] = sourse[i]; //put it in destinate
        k = count;
    count++;
  }
  if(count>=size) return;

  i++;
  if(i>=size) i = 1;
  if(sourse[i].idx>0 && destinate[k].id==sourse[i].pnt)
   continue;
  else
  {
   start = i;
   do
   {
    reset = search(sourse, start, size, destinate[k].id);
    k--;
    start = 1;
   } while(reset < 0 && k >= 0);

   if(reset>=0) i = reset;
  }

 } while(count < size);
}
//==========================================================================

int search(vtx *mst, int begin, int size, int id)
{
 int i;

  for(i=begin; i<size; i++)
  {
   if(mst[i].idx>0 && mst[i].pnt==id) //here need new node
   return i;
  }

 return (-1);
}
//==========================================================================
```

Program for implementing the New1 algorithm

```
float exch(vtx *que, vtx *adt, vtx *mst, int size, float cos) //cos is the upper bound
{
 int q1h, q1t, q2h, q2t, q3h, q3t, q4h, q4t;
 float seq1, seq2, seq3, seq4;
 vtx q1[2], q2[2], q3[2], q4[2];
 int *order;
 float sum, sub, upbound;
 int i, j, mode, flag;

 order = (int *) calloc (size, sizeof(int));
 if (order == NULL)
 {
   printf("out of space\n");
   exit (0);
 }

 flag = 4;          //initiating the number of exchanged edges
 upbound = cos;  //initiating the upper bound as original length

while(flag<=4)  //flag>4 means no improvement can get, exit loop
{
  //call setbreak() to break 3 or 4 longest edges and return break points
 setbreak(que, q1, q2, q3, q4, flag, size);

  //call symble() to set all break point's position
 symble(q1,q2,q3,q4,&q1t,&q1h,&q2t,&q2h,&q3t,&q3h,&q4t,&q4h,flag);

 for (i=0; i<size; i++)       //copy the new order in adt
  adt[i] = que[i];

 seq1 = count(adt, q1h, q1t, size);  //count every sequence length
 if (seq1<0.00001) //there is a single break node
 {  seq1 = insertsingle(que, adt, q1h, size);
         if (seq1>0.00001 && seq1<upbound) //insert the single node successfully
         {          upbound = seq1;
                    continue;
         }
 }
 seq2 = count(adt, q2h, q2t, size);
 if (seq2<0.00001) //there is a single break node
 {  seq2 = insertsingle(que, adt, q2h, size);
         if (seq2>0.00001 && seq2<upbound) //insert the single node successfully
         {          upbound = seq2;
                    continue;
         }
 }
 seq3 = count(adt, q3h, q3t, size);
 if (seq3<0.00001) //there is a single break node
 {  seq3 = insertsingle(que, adt, q3h, size);
         if (seq3>0.00001 && seq3<upbound) //insert the single node successfully
         {          upbound = seq3;
                    continue;
         }
 }
 if (flag==4)
 {
         seq4 = count(adt, q4h, q4t, size);
   if (seq4<0.00001)  //there is a single break node
         {  seq4 = insertsingle(que, adt, q4h, size);
                    if (seq4>0.00001 && seq4<upbound)  //insert the single node successfully
                    {           upbound = seq4;
                                continue;
                    }
         }
 }

   //if everything is ok, count all possible edge-switch cost
 mode = switchcost(q1, q2, q3, q4, &sub, flag);

 sum = seq1 + seq2 + seq3 + sub;
 if (flag==4)
    sum = sum + seq4;

         //float can not compare exactly so I use error rate to control the loop
 if ((upbound-sum)<=0.00001 || mode==1)
 {
   flag++;
   continue;
 }

 if (flag==4)
     getorder4(q1t, q2h, q2t, q3h, q3t, q4h, q4t, q1h, size, mode, order);
 else
     getorder3(q1t, q2h, q2t, q3h, q3t, q1h, size, mode, order);

 for (i=0; i<size; i++)        //record the new order in que
   {
     j = order[i];
     que[i] = adt[j];
   }

 upbound = totalcost(que, size);  //critical step, change upper bound

                                  //if (flag==4) flag--;  //back to 3-opt mode

} //the end of main while loop (flag<=4)

free(order);
return upbound;
}
//=========================================================================

void getorder4(int q1t,int q2h,int q2t,int q3h,int q3t,int q4h,int q4t, int q1h,  int size, int mode, int *order)
{
 switch (mode)
 {
  case 1:
           setorder4(q1t, q2h, q2t, q3h, q3t, q4h, q4t, q1h, size, order); break;
  case 2:
           setorder4(q1t, q2h, q2t, q3h, q3t, q4t, q4h, q1h, size, order); break;
```

```
case 3:
        setorder4(q1t, q2h, q2t, q3t, q3h, q4h, q4t, q1h, size, order); break;
case 4:
      setorder4(q1t, q2h, q2t, q3t, q3h, q4t, q4h, q1h, size, order); break;
case 5:
      setorder4(q1t, q2t, q2h, q3h, q3t, q4h, q4t, q1h, size, order); break;
case 6:
      setorder4(q1t, q2t, q2h, q3h, q3t, q4t, q4h, q1h, size, order); break;
case 7:
      setorder4(q1t, q2t, q2h, q3t, q3h, q4h, q4t, q1h, size, order); break;
case 8:
      setorder4(q1t, q2t, q2h, q3t, q3h, q4t, q4h, q1h, size, order); break;
case 9:
      setorder4(q1t, q4t, q4h, q3t, q3h, q2t, q2h, q1h, size, order); break;
case 10:
      setorder4(q1t, q4h, q4t, q3t, q3h, q2t, q2h, q1h, size, order); break;
case 11:
      setorder4(q1t, q4t, q4h, q3h, q3t, q2t, q2h, q1h, size, order); break;
case 12:
      setorder4(q1t, q4h, q4t, q3h, q3t, q2t, q2h, q1h, size, order); break;
case 13:
      setorder4(q1t, q4t, q4h, q3t, q3h, q2h, q2t, q1h, size, order); break;
case 14:
      setorder4(q1t, q4h, q4t, q3t, q3h, q2h, q2t, q1h, size, order); break;
case 15:
      setorder4(q1t, q4t, q4h, q3h, q3t, q2h, q2t, q1h, size, order); break;
case 16:
      setorder4(q1t, q4h, q4t, q3h, q3t, q2h, q2t, q1h, size, order); break;
case 17:
      setorder4(q1t, q2h, q2t, q4h, q4t, q3h, q3t, q1h, size, order); break;
case 18:
      setorder4(q1t, q2h, q2t, q4h, q4t, q3t, q3h, q1h, size, order); break;
case 19:
      setorder4(q1t, q2h, q2t, q4t, q4h, q3h, q3t, q1h, size, order); break;
case 20:
      setorder4(q1t, q2h, q2t, q4t, q4h, q3t, q3h, q1h, size, order); break;
case 21:
      setorder4(q1t, q2t, q2h, q4h, q4t, q3h, q3t, q1h, size, order); break;
case 22:
      setorder4(q1t, q2t, q2h, q4h, q4t, q3t, q3h, q1h, size, order); break;
case 23:
      setorder4(q1t, q2t, q2h, q4t, q4h, q3h, q3t, q1h, size, order); break;
case 24:
      setorder4(q1t, q2t, q2h, q4t, q4h, q3t, q3h, q1h, size, order); break;
case 25:
      setorder4(q1t, q3t, q3h, q4t, q4h, q2t, q2h, q1h, size, order); break;
case 26:
      setorder4(q1t, q3h, q3t, q4t, q4h, q2t, q2h, q1h, size, order); break;
case 27:
      setorder4(q1t, q3t, q3h, q4h, q4t, q2t, q2h, q1h, size, order); break;
case 28:
      setorder4(q1t, q3h, q3t, q4h, q4t, q2t, q2h, q1h, size, order); break;
case 29:
      setorder4(q1t, q3t, q3h, q4t, q4h, q2h, q2t, q1h, size, order); break;
case 30:
      setorder4(q1t, q3h, q3t, q4t, q4h, q2h, q2t, q1h, size, order); break;
```

```
   case 31:
          setorder4(q1t, q3t, q3h, q4h, q4t, q2h, q2t, q1h, size, order); break;
   case 32:
          setorder4(q1t, q3h, q3t, q4h, q4t, q2h, q2t, q1h, size, order); break;
   case 33:
          setorder4(q1t, q3h, q3t, q2h, q2t, q4h, q4t, q1h, size, order); break;
   case 34:
          setorder4(q1t, q3h, q3t, q2h, q2t, q4t, q4h, q1h, size, order); break;
   case 35:
          setorder4(q1t, q3h, q3t, q2t, q2h, q4h, q4t, q1h, size, order); break;
   case 36:
          setorder4(q1t, q3h, q3t, q2t, q2h, q4t, q4h, q1h, size, order); break;
   case 37:
          setorder4(q1t, q3t, q3h, q2h, q2t, q4h, q4t, q1h, size, order); break;
   case 38:
          setorder4(q1t, q3t, q3h, q2h, q2t, q4t, q4h, q1h, size, order); break;
   case 39:
          setorder4(q1t, q3t, q3h, q2t, q2h, q4h, q4t, q1h, size, order); break;
   case 40:
          setorder4(q1t, q3t, q3h, q2t, q2h, q4t, q4h, q1h, size, order); break;
   case 41:
          setorder4(q1t, q4t, q4h, q2t, q2h, q3t, q3h, q1h, size, order); break;
   case 42:
          setorder4(q1t, q4h, q4t, q2t, q2h, q3t, q3h, q1h, size, order); break;
   case 43:
          setorder4(q1t, q4t, q4h, q2h, q2t, q3t, q3h, q1h, size, order); break;
   case 44:
          setorder4(q1t, q4h, q4t, q2h, q2t, q3t, q3h, q1h, size, order); break;
   case 45:
          setorder4(q1t, q4t, q4h, q2t, q2h, q3h, q3t, q1h, size, order); break;
   case 46:
          setorder4(q1t, q4h, q4t, q2t, q2h, q3h, q3t, q1h, size, order); break;
   case 47:
          setorder4(q1t, q4t, q4h, q2h, q2t, q3h, q3t, q1h, size, order); break;
   case 48:
          setorder4(q1t, q4h, q4t, q2h, q2t, q3h, q3t, q1h, size, order); break;
   default:;
  }
}
//==========================================================================

void setorder4(int p1,int p2,int p3,int p4,int p5,int p6,int p7,int p8,int size,int *ary)
{
 int i;

 ary[0] = p1;
 for (i=1; i<size; i++)
 {
   if (p2>p3)
    ary[i] = p2--;
   else if (p2<p3)
    ary[i] = p2++;
   else
   {
    ary[i] = p2;
    break;
    }
  }
  for (++i; i<size; i++)
  {
    if (p4>p5)
     ary[i] = p4--;
    else if (p4<p5)
     ary[i] = p4++;
    else
    {
     ary[i] = p4;
     break;
    }
  }
  for (++i; i<size; i++)
  {
    if (p6>p7)
     ary[i] = p6--;
    else if (p6<p7)
     ary[i] = p6++;
    else
    {
     ary[i] = p6;
     break;
    }
  }
  for (++i; i<size; i++)
  {

    ary[i] = p8%size;
            p8++;
  }
}
//====================================================================

void setbreak (vtx *que,vtx *q1,vtx *q2,vtx *q3,vtx *q4,int flag,int size)
{
  float larg1, larg2, larg3, larg4;
  int break1, break2, break3, break4, i;
  int brak[4] = {-1, -1, -1, -1};

  larg1 = larg2 = larg3 = larg4 = 0.0;
  break1 = break2 = break3 = break4 = 99999999;

  for(i=0; i<size; i++)
  {
    if(que[i].cost>larg1)
    {
      larg4 = larg3;
      brak[3] = brak[2];
      larg3 = larg2;
      brak[2] = brak[1];
      larg2 = larg1;
           brak[1] = brak[0];
      larg1 = que[i].cost;
           brak[0] = i;
      continue;
    }
    else if(que[i].cost>larg2)
    {
      larg4 = larg3;
      brak[3] = brak[2];
      larg3 = larg2;
      brak[2] = brak[1];
      larg2 = que[i].cost;
      brak[1] = i;
      continue;
    }
    else if(que[i].cost>larg3)
    {
      larg4 = larg3;
      brak[3] = brak[2];
      larg3 = que[i].cost;
      brak[2] = i;
      continue;
    }
    else if(que[i].cost>larg4)
    {
      larg4 = que[i].cost;
      brak[3] = i;
      continue;
    }
    else ;
  }


  for(i=0; i<flag; i++)
  {
    if(brak[i]<break1)
    {
      break4 = break3;
      break3 = break2;
      break2 = break1;
      break1 = brak[i];
      continue;
    }
    else if(brak[i]<break2)
    {
      break4 = break3;
      break3 = break2;
      break2 = brak[i];
      continue;
    }
    else if(brak[i]<break3)
    {
      break4 = break3;
      break3 = brak[i];
      continue;
    }
    else if(brak[i]<break4)
    {
      break4 = brak[i];
      continue;
    }
    else ;
  }

  q1[0] = que[break1];
  q1[0].idx = break1;
  q2[0] = que[break1+1];
  q2[0].idx = break1+1;
  q2[1] = que[break2];
  q2[1].idx = break2;
  q3[0] = que[break2+1];
  q3[0].idx = break2+1;
  q3[1] = que[break3];
  q3[1].idx = break3;
  if(flag==4)
  {
   q4[0] = que[break3+1];
   q4[0].idx = break3+1;
   q4[1] = que[break4];
   q4[1].idx = break4;
   q1[1] = que[(break4+1)%size];
   q1[1].idx = (break4+1)%size;
  }
  else
  {
   q1[1} = que[(break3+1)%size];
   q1[1].idx = (break3+1)%size;
  }

}

//=============================================================================

int switchcost(vtx *q1, vtx *q2, vtx *q3, vtx *q4, float *sub, int flag)
{
  int l, m, n, p, index, mode;
  float sum, sb;

  index = 1;   //initiating the value for mode
  sb = 9999999.0;  //initiating the value for record sub length
  sum = 0.0; //for counting various combination

  for(l=0; l<2; l++) //fiqure out the shortest combination of edges
  {
          for(m=0; m<2; m++)
          {
                  for(n=0; n<2; n++)
                  {
                   if(flag==4)
                   {
                     for(p=0; p<2; p++)
                     {
                             sum += cost(q1[l].x, q1[l].y, q2[m].x, q2[m].y);
                             sum += cost(q2[m^1].x, q2[m^1].y, q3[n].x, q3[n].y);
                             sum += cost(q3[n^1].x, q3[n^1].y, q4[p].x, q4[p].y);
```

```
                    sum += cost(q4[p^1].x, q4[p^1].y, q1[l^1].x, q1[l^1].y);
         if (sum < sb)
         {
           sb = sum;
           mode = index;
         }
         sum = 0.0;

                        sum += cost(q1[l].x, q1[l].y, q2[m].x, q2[m].y);
                    sum += cost(q2[m^1].x, q2[m^1].y, q4[n].x, q4[n].y);
                    sum += cost(q4[n^1].x, q4[n^1].y, q3[p].x, q3[p].y);
                    sum += cost(q3[p^1].x, q3[p^1].y, q1[l^1].x, q1[l^1].y);
         if (sum < sb)
         {
           sb = sum;
           mode = index+16;
         }
         sum = 0.0;

                        sum += cost(q1[l].x, q1[l].y, q3[m].x, q3[m].y);
                    sum += cost(q3[m^1].x, q3[m^1].y, q2[n].x, q2[n].y);
                    sum += cost(q2[n^1].x, q2[n^1].y, q4[p].x, q4[p].y);
                    sum += cost(q4[p^1].x, q4[p^1].y, q1[l^1].x, q1[l^1].y);
         if (sum < sb)
         {
           sb = sum;
           mode = index+32;
         }
         sum = 0.0;

                        index += 1;

              }
             }
       else
             {
        sum += cost(q1[l].x, q1[l].y, q2[m].x, q2[m].y);
        sum += cost(q2[m^1].x, q2[m^1].y, q3[n].x, q3[n].y);
        sum += cost(q3[n^1].x, q3[n^1].y, q1[l^1].x, q1[l^1].y);
        if (sum < sb)
        {
          sb = sum;
          mode = index;
        }
        sum = 0.0;
        index += 1;
             }
            }
       }
  }

  *sub = sb;
  return mode;

}
//=================================================================
float insertsingle (vtx *que, vtx *origi, int start, int size)
{
 int i, j, k, position;
 float upbound, limit, sumcos, length;
 vtx temp;

 if (start==0)
 { j = size-1;
  k = 1;
 }
 else if (start==(size-1))
 { k = 0;
  j = start-1;
 }
 else
 { j = start-1;
  k = start+1;
 }

 upbound = cost(origi[start].x, origi[start].y, origi[j].x, origi[j].y) +
                              cost(origi[start].x, origi[start].y, origi[k].x, origi[k].y) -
                              cost(origi[j].x, origi[j].y, origi[k].x, origi[k].y);

 limit = upbound;

 if (start==0)
 {       k = 2;
         j = 1;
 }
 else if (start==(size-1))
 { k = 1;
         j = 0;
 }
 else
 { j = start+1;
         k = j+1;
         if(k>=size) k=0;
 }

 for(i=0; i<size-2; i++)
 {

                 sumcos = 0.0;
                 if(j>=size) j=0;
                 if(k>=size) k=0;
                 length = cost(origi[start].x, origi[start].y, origi[j].x, origi[j].y);
                 sumcos += length;
     length = cost(origi[start].x, origi[start].y, origi[k].x, origi[k].y);
                 sumcos += length;
     length = cost(origi[j].x, origi[j].y, origi[k].x, origi[k].y);
                 sumcos -= length;

             if ((upbound-sumcos)>0.00001)
                 {
                  upbound = sumcos;
```

```
                        position = k;  //the biggest position will be equal to count
                    }
                    j++; k++;
  }

  if((limit-upbound)<0.00001)        //that means no improving
          return 0.0;

  temp = origi[start];
  k = start;
  if (position>start)
  { for(; k<position; k++)
                  origi[k] = origi[k+1];
  }
  if (position<start)
  { for(; k>position; k--)
                  origi[k] = origi[k-1];
  }
  origi[k] = temp;  //insert the single node into the shortest position

  for (i=0; i<size; i++)      //record the new order in que
     que[i] = origi[i];

  upbound = totalcost(que, size);  //critical step, change upbound
  return upbound;
}
//==================================================================
```

Program for implementing the New 2 algorithm

```
void insertion (vtx *que, vtx *origi, int size)

{
  int count, i, j, k, position;
  float upbound, sumcos, length;

  for(count=0; count<3; count++)
     que[count] = origi[count];

   while(count<size)        //that is O(n^2) algorithm
   {
    upbound = 9999999.0;
          for(j=0; j<count; j++)
          {
                  sumcos = 0.0;
                  i = j+1;
                  if(i==count) i=0;
                  length = cost(origi[count].x, origi[count].y, que[j].x, que[j].y);
                  sumcos += length;
      length = cost(origi[count].x, origi[count].y, que[i].x, que[i].y);
                  sumcos += length;
      length = cost(que[j].x, que[j].y, que[i].x, que[i].y);
                  sumcos -= length;

             if (sumcos<upbound)
                  {
                    upbound = sumcos;
                    position = j+1;  //the biggest position will be equal to count
                   }
            }
            k = count;
            while(k>position)  //insert the new node into the shortest position
            {
                    que[k] = que[k-1];
                    k--;
            }
            que[k] = origi[count];
            count++;
          } //the end of while(count<size)
}
//=====================================================================
```

Program for implementing the random 3-opt algorithm

```
//implementation 3-opt algorithm for TSP tour and used for VB interface
//using random start point to do exchange edges
//revised count() and setorder(), do 0.45sqrt(n)*n switching to implementation the tour

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include "common.h"

#define TOUR 0
#define RAND 1

void setstart(vtx *dester, vtx *source, int j);
float count(vtx *adt, int a, int b, int size);
float bell(vtx *que, vtx *adt, int sep1, int sep2, int start, int size, float cos);
void setorder(int p1,int p2,int p3,int p4,int p5,int p6,int size,int *ary);

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

int main(int argc, char * argv[])
{
  int i, j, k, l, m, n, repeat, nochange, flag, size;
  float sumcos, aversum, longest, shortest;
  char title1[80] = "========== Running Results of 3-Opt Algorithm ==========";
  char title2[80] = "========== Paths of the Resulted 3-Opt Tour ==========";

  char origcity[30] = "";
  char opttour[30] = "C:\\wxiaozh\\opttour";
  char optsum[30] = "C:\\wxiaozh\\optsum";
  char optdata[30] = "C:\\wxiaozh\\optdata";

  vtx *origi;
  vtx *adt;
  vtx *que;

  if (argc<3)
  {
```

```
  printf("Running command should be as following example: \n");
  printf("execution-file input-file number-of-vertex [number-of-iterate]\n");
  printf("  *.exe      inputfile    integer       [interger] \n");
        exit (0);
 }

 j = atoi(argv[2]);
 strcpy(origcity, argv[1]);
 n = 1;
 if (argc>3) n = atoi(argv[3]); //if fourth argument, get the number of iterate

 repeat = n;
 nochange = 0;
 flag = 1;
 longest = 0.0;
 shortest = 999999.0;
 aversum = 0.0;

 origi = dynmem(j); //set memory for computing
 que = dynmem(j);
 adt = dynmem(j);

 while(n>0) //this loop is for iterate times of executing
 {
  if (n == 1) //set vertex array with designed input data file
    setvertices(origcity, origi, j);
  else  //get input data from program's random function
        setvertex(origi, j);

  for(i=0; i<j; i++)
   que[i]=origi[i];

                 //the optimal formula to get the value of k
 //k = (int)(sqrt(j)*4.5);

         k = j;  // running time will be j square
  sumcos = 999999.0;

  while(k--)  //k is the times to change the start vertex of the graph so
  {        //that new cut-swap can be used
   setstart(adt, que, j);

   size = (int)(sqrt(j)*4.5);
         //size = j;
   while(size--)  //size is the times to do cut-swap when the start vertex
   {          //is not changed. The total complexity is k squared.
    while(1)
    {
     m = (int)(urand()*1000)%j;   //get a random cutting point of array
     l = (int)(urand()*1000)%j;    //get a random cutting point of array
     if(abs(m-l)>2 && m>1 && l>1)
       break;
    }

    sumcos = bell(que, adt, m, l, 0, j, sumcos);  //critical function doing 3-opt
    }      //while(size--) end
   }      //while(K--) end

          aversum = aversum + sumcos;
  if(sumcos>longest) longest = sumcos;
  if(sumcos<shortest) shortest = sumcos;

  n--;

  } //end of while (n>0)

  nochange = -1;
  aversum = aversum/repeat;
  writevertices(opttour, que, j); //write an data file of all vertices coordinates
  printdata(optsum, title1, sumcos, aversum, longest, shortest, repeat, nochange, j);
  #if TOUR
   printpath(optdata, title2, que, j); //write the whole tour of the MST algorithm
  #endif

  free(origi);
  free(que);
  free(adt);

} //end of main


//=====================================================================

void setstart(vtx *dester, vtx *source, int j)
{
  int start, f, idx;

  start = (int)(urand()*1000)%j;      //get a random start point
#if RAND         //if random starting mode, change the first node
  f = 0;
  idx = 0;
#else    //if not random starting mode, change only from second node
  while(start==0)
   start = (int)(urand()*1000)%j;    //here start should be !=0
  f = 1;
  idx = 1;
  dester[0] = source[0]; //always keep the first node non-change
#endif

  for(; f<j; f++)
  {
    dester[f] = source[start++];
    if(start>=j)
      start = idx;
  }
}

float bell(vtx *que, vtx *adt, int sep1, int sep2, int start, int size, float cos)
{
  int q1h, q1t, q2h, q2t, q3h, q3t;
  float sum, seq1, seq2, seq3;
```

```
vtx q1[2], q2[2], q3[2];
float sub = 999999.0;      //initiating the value for record sub length
int index = 1;    //initiating the value for mode
int i, j, mode, l, m, n;
int *order;

q1t = start;
q2h = start + 1;
q2t = (sep1<sep2)?sep1:sep2;
q3h = q2t + 1;
q3t = (sep1>sep2)?sep1:sep2;
q1h = q3t + 1;

q1[0] = adt[q1t];
q1[1] = adt[q1h];
q2[0] = adt[q2h];
q2[1] = adt[q2t];
q3[0] = adt[q3h];
q3[1] = adt[q3t];

order = (int *) calloc (size, sizeof(int));
if (order == NULL)
 {
 printf("can not get space for order array.\n");
 exit (0);
 }

seq1 = count(adt, q1h, q1t, size);  //count every sequence length
seq2 = count(adt, q2h, q2t, size);
seq3 = count(adt, q3h, q3t, size);
sum = 0.0;

for(l=0; l<2; l++)
{
          for(m=0; m<2; m++)
          {
                    for(n=0; n<2; n++)
                    {
      sum += cost(q1[l].x, q1[l].y, q2[m].x, q2[m].y);
      sum += cost(q2[m^1].x, q2[m^1].y, q3[n].x, q3[n].y);
      sum += cost(q3[n^1].x, q3[n^1].y, q1[l^1].x, q1[l^1].y);
      if (sum < sub)
      {
       sub = sum;
       mode = index;
      }
      sum = 0.0;
      index += 1;
                    }
          }
}

sum = seq1 + seq2 + seq3 + sub;
if(sum>cos)
  return cos;
 switch (mode)
 {
  case 1:
      setorder(q1t, q2h, q2t, q3h, q3t, q1h, size, order); break;
  case 2:
      setorder(q1t, q2h, q2t, q3t, q3h, q1h, size, order); break;
  case 3:
      setorder(q1t, q2t, q2h, q3h, q3t, q1h, size, order); break;
  case 4:
      setorder(q1t, q2t, q2h, q3t, q3h, q1h, size, order); break;
  case 5:
      setorder(q1t, q3t, q3h, q2t, q2h, q1h, size, order); break;
  case 6:
      setorder(q1t, q3h, q3t, q2t, q2h, q1h, size, order); break;
  case 7:
      setorder(q1t, q3t, q3h, q2h, q2t, q1h, size, order); break;
  case 8:
                    setorder(q1t, q3h, q3t, q2h, q2t, q1h, size, order); break;
  default:;
 }

 for (i=0; i<size; i++)       //record the new order in que
 {
  j = order[i];
  que[i] = adt[j];
 }

 for (i=0; i<size; i++)       //copy the new order in adt
  adt[i] = que[i];

 free(order);
 return sum;
}
//==========================================================================
```

Program for implementing the random insertion algorithm

```
void insertion(vtx *origi, vtx *que, int size)
{
 int count, i, j, k, position;
 float upbound, sumcos, length;

 for(count=0; count<3; count++)
    que[count] = origi[count];

   while(count<size)        //follows maybe not right! Check it.
   {
    upbound = 9999999.0;
          for(j=0; j<count; j++)
          {
                    sumcos = 0.0;
                    i = j+1;
                    if(i==count) i=0;
                    length = cost(origi[count].x, origi[count].y, que[j].x, que[j].y);
                    sumcos += length;
      length = cost(origi[count].x, origi[count].y, que[i].x, que[i].y);
```

```
                    sumcos += length;
        length = cost(que[j].x, que[j].y, que[i].x, que[i].y);
                    sumcos -= length;

                if (sumcos<upbound)
                    {
                     upbound = sumcos;
                     position = j+1;
                    }
            }
            k = count;
            while(k>position)
            {
                    que[k] = que[k-1];
                    k--;
            }
            que[k] = origi[count];
            count++;
           } //the end of while(count<size)
}
```

VITA

Xiaozhen Wang

Candidate for the Degree of

Master of Science

Thesis: MODIFIED MST ALGORITHMS FOR TSP TOURS

Major Field: Computer Science

Biographical:

Personal Data: Born in Shanghai, China, the son of Chongdao Wang and Wenqin Zhang.

Education: Received Bachelor of Law degree in Political science and a Master of Law in Sociology from East China Normal University, Shanghai, China in July 1982 and July 1985, respectively. Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in May 2000.