ANIMATION OF APPROXIMATION ALGORITHMS FOR TRAVELING SALESMAN PROBLEM IN JAVA

By

YAN GU

Bachelor of Science
Nankai Unoversity
Tianjin, P. R. China
1991

Master of Science
Nankai University
Tianjin, P. R. China
1994

Master of Science
Oklahoma State University
Stillwater, Oklahoma
1997

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
December 1998

ANIMATION OF APPROXIMATION ALGORITHMS FOR

TRAVELING SALESMAN PROBLEM IN JAVA

Thesis Approved:

J Chandler
Thesis Advisor

[illegible]

M. Samadzadeh

Wayne B. Powell
Dean of the Graduate College

ACKNOWLEDGMENTS

I wish to express my sincere appreciation to my major advisor, Dr. J. P. Chandler, for his intelligent supervision, constructive guidance, inspiration, encouragement, and friendship.

Sincere thanks also go to my committee members, Dr. M. H. Samadzadeh and Dr. G. E. Hedrick, whose guidance, instruction, encouragement, and friendship are also invaluable.

I would also like to express my appreciation to my husband, Tao Zhu, for his encouragement, love, and understanding through this whole process. Thanks also go to my parents for their support and encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

NOMENCLATURE

TSP	Traveling Salesman Problem
VLSI	Very Large Scale Integration
GUI	Graphical User Interface
WWW	World Wide Web
NN	Nearest Neighbor algorithm
MST	Minimum Spanning Tree
JVM	Java Virtual Machine
HTML	Hypertext Makeup Language
HTTP	Hypertext Transfer Protocol
URL	Uniform or Universal Resource Locator
AWT	Abstract Window Toolkit

CHAPTER I

INTRODUCTION

The Traveling Salesman Problem (TSP) is a classic problem in combinatorial optimization. The TSP is easy to state; however it is quite challenging to find a good optimization algorithm or an approximation algorithm that is guaranteed to be effective.

1.1 Definition of Traveling Salesman Problem

The definition of the Traveling Salesman Problem is:

Given: a set of cities $C_1, C_2 \ldots C_n$, and a distance $d\ (C_i, C_j)$ for each distinct pair of cities, C_i and C_j.

Find: a permutation P of the cities that minimizes the tour length of a traveling salesman who visits the cities in the order specified by the permutation, and returns to the initial city at the end.

The tour length is: $\sum_{i=1}^{N-1} d\left(C_{P(i)}, C_{P(i+1)}\right) + d\left(C_{P(n)}, C_{P(1)}\right)$.

A TSP is said to be symmetrical if and only if the distances between each pair of distinct cities satisfy $d\ (c_i, c_j) = d\ (c_j, c_i)$ for $1 \leq i, j \leq n$. Also, a TSP is said to satisfy the triangle inequality if and only if $d\ (c_i, c_j) + d\ (c_j, c_k) \geq d\ (c_i, c_k)$, where $1 \leq i, j, k \leq n$.

1.2 The Application of Traveling Salesman Problem

The symmetric traveling salesman problem has many applications, including computer wiring [Lenstra and Rinnooy Kan 1975], wallpaper cutting [Garfinkel 1977], hole punching [Reinelt 1989], job sequencing, Very Large Scale Integration (VLSI) chip fabrication [Korte 1989], X-ray crystallography [Bland & Shallcross 1989], and dartboard design [Eiselt and Laporte 1991].

In addition, many new algorithmic ideas were developed in the ground of TSP. Important early works on branch and bound algorithms, cutting plane techniques and local optimization all utilized the TSP as the initial proving ground. Furthermore, the TSP was one of the first problems that the new theory of NP-completenes was applied to in the early 1970s, and it has been commonly referred as the classic example of an NP-hard combinatorial optimization problem [Johnson 1990].

1.3 The Objective of the Thesis

The TSP problem is a good example of algorithm design and problem solving. People may get dramatic benefits from learning the various TSP algorithms. Since the TSP is an NP-hard problem, which means that it is not likely to find the optimal tours in deterministic polynomial-time, much research has been focussed on efficient approximation algorithms, which are fast algorithms whose attempt is to find a near-optimal rather than an optimal tour.

The objective of this thesis is to design and implement a software program that uses a Graphical User Interface (GUI) to animate some of the most popular approximation algorithms developed so far for the symmetric TSP, and to display a graphical running result for each algorithm. This software is aimed to help people learn the algorithms easily and get a deeper understanding of them. The program is designed to run either as a standalone application or on the World Wide Web (WWW), so that it allows more people to get access and benefit from it easily. Java is regarded as a very good language for writing programs on the WWW, and it also has a lot of good features as a programming language compared to other languages. Therefore, Java was chosen as the programming language in which to implement this project.

Chapter II is a literature review of the related components used in this project. Chapter III gives the details of the design and implementation of this project. Some example snapshots of the running results are illustrated in Chapter IV. Finally, the summary and future work recommendations are given in Chapter V. Also, a glossary is appended at the end.

CHAPTER II

LITERATURE REVIEW

This project uses Java as the programming language, and the WWW as the medium, to animate some approximation algorithms for the Traveling Salesman Problem. The following are reviews of some of the thesis components.

2.1 Approximation Algorithms for Traveling Salesman Problem

Since the TSP is an NP-hard problem, it is natural to try to solve it by means of heuristic algorithms. The TSP heuristics currently most often covered in the computer science field are the *tour construction heuristics* and the *local optimization heuristics*. Tour construction heuristics gradually build up a tour out of shorter paths or cycles. Local optimization heuristics make local improvements to existing tours.

2.1.1 Tour Construction Heuristics

Nearest Neighbor Algorithm [Rosenkrantz et al. 1977]. The most natural heuristic to solve the TSP is the Nearest Neighbor (*NN*) algorithm. In this algorithm, the traveler starts from an arbitrarily chosen initial city, then repeatedly chooses the next city which is unvisited and closest to the current one. Once all cities have been visited, the traveler returns to the initial city to close the tour. The reason that this algorithm draws much attention is its simplicity.

Step 1. Arbitrarily choose a vertex as the initial city.

Step 2. Find the closest unvisited vertex to the current vertex to visit next. Repeat Step 2, until all vertices have been visited.

Step 3. Link the last vertex of the tour to the first one.

The complexity of this procedure is $O(n^2)$. A possible modification is to apply the above procedure n times with n different vertices as the starting point, then choose the shortest tour as the final result. The overall algorithm complexity is then $O(n^3)$, but the tour generated by this modification *NN* algorithm is generally better. It can be guaranteed that $NN\ (I)\ /\ OPT\ (I) \leq 0.5\ (\lfloor log_2\ (n) \rfloor + 1)$, where $NN\ (I)$ is the length of the tour constructed by the Nearest Neighbor algorithm over instance I, and $OPT\ (I)$ is the optimum tour length over instance I [Rosenkrantz et al. 1977].

Greedy Algorithm [Benltly and Saxe 1980]. Some people refer to the Nearest Neighbor Algorithm as Greedy, but it is more appropriately to call the following algorithm a Greedy algorithm. This heuristic is to consider the edges of the graph in the order of non-decreasing length, and add an edge to the tour whenever to do so will create neither a vertex whose degree exceeds two nor a cycle with fewer than n cities.

Step 1. Start with the shortest edge.

Step 2. Add the shortest remaining edge to the tour, if adding it would not create a vertex with degree more than two or a cycle of length less than n. Repeat Step 2 until all cities are visited.

The time complexity for this Greedy algorithm is $O(n^2\ log\ (n))$ and thus it is slightly slower than Nearest Neighbor Algorithm. However, its worst-case quality of the tour may be somewhat better in terms of tour length. It can be shown that

Greedy (I) / *OPT* $(I) \leq 0.5\ (\lceil \log_2 (n) \rceil + 1)$, where *Greedy* (I) is the length of the tour constructed by the Greedy algorithm over instance I, and *OPT* (I) is the optimum tour length over instance I [Ong. and Moore 1984].

Insertion Heuristics [Rosenkrantz, et al. 1977]. There are several similar algorithms that belong to the Insertion Heuristics. All of them share the same basic structure, but each has different selection rules.

Step 1. Select two vertices consisted in the starting tour.

Step 2. Select an unvisited vertex; insert it in the proper position of the tour so that the insertion causes minimum increase of the total tour length. If any vertex has not been considered, repeat Step 2.

There are 3 kinds of selection rules to select the next vertex to be inserted in the tour.

a) Nearest Insertion --- Start from the two closest vertices, and the next vertex to be included in the tour is the closest one to the current tour.

b) Farthest Insertion --- Start from the two vertices that are farthest apart, and the vertex farthest away from the current tour is the one to be inserted next.

c) Random Insertion --- Randomly select the starting tour and the next vertex to insert.

The complexity of various Insertion Heuristics lies between $O(n^2)$ and $O(n\ log\ (n))$, depending on which selection rule it is used. It has been shown that the length of any tour constructed by the Insertion Heuristics is never more than $\lceil \log_2 (n) \rceil + 1$ times the length of the optimal tour [Rosenkrantz et al. 1977]. The length of the Nearest Insertion tour is no more than twice that of the optimal tour [Rosenkrantz et al. 1977] and the length of the Farthest Insertion tour is no more than 3/2 times of the length of the optimal tour [Johnson and Papadimitriou, 1985].

Christofides Algorithm [Christofides 1976]. Christofides algorithm makes sophisticated use of the minimum spanning tree (MST). A minimum spanning tree is an acyclic subset of the edges from an undirected graph, which connects all of the vertices, and whose total weight is minimized [Weiss 1997].

Step 1. Construct a minimum spanning tree (MST) on the set of all cities.

Step 2. Construct a minimum cost matching of odd degree vertices in the MST. Combine the matching edges with the MST edges.

Step 3. Construct a tour from the graph.

In this algorithm, the construction of the MST takes $O(n^2)$ time, and the computation of a minimum cost matching can be done in $O(n^3)$ time [Lawler 1976]. Now an Euler cycle must exist in this graph. The Euler cycle is a simple cycle that passes through each edge exactly once. The traveling salesman tour can be constructed in $O(n)$ time, by adding the cities into the tour, in the order when they are first encountered while traversing the Euler cycle. The running time for this algorithm is dominated by the time for finding the minimum cost matching in Step 2, hence is $O(n^3)$.

The worst-case length of the tour constructed by Christofides algorithm is no more than 3/2 that of the optimal tour, assuming the triangle inequality is obeyed [Johnson and McGeoch 1997]. Thus its worst-case tour length is the shortest among all currently known tour construction heuristics. However, Christofides Algorithm is considerably slower than other tour construction heuristics. This is mainly because the best available matching algorithm takes $O(n^3)$ time. Due to the substantial programming effort needed, nobody has ever actually implemented this algorithm [Johnson and McGeoch 1997]. In this project, the same approximation algorithm used in constructing MST is employed to implement the minimum cost matching step.

Strip Algorithm [Chandler 1998]. The Strip algorithm was invented by P. Yager of University of California at San Diego in the early 1960's. The Strip algorithm is also referred as "The Boustrophedonic (As The Ox Plows) Algorithm" by Jon Bentley. For simplicity, all points are assumed to be confined to the unit square.

Step 1. Divided the unit square into $\sqrt{n}$ equal-width strips.

Step 2. Start from the leftmost strip, proceed through the strips one by one from left to right, visiting the cities within each strip in the alternatively descending and ascending directions.

The Strip algorithm is extremely efficient: Step 1 takes $O(n)$ time, and the running time for this algorithm is dominated by Step 2, which takes $O(n \log(n))$ time.

2.1.2 Local Optimization Heuristics

The tour constructed by the above tour construction algorithms can be further improved by the local optimization heuristics. The basic structure of the local optimization heuristics is:

Step 1. Construct a starting tour, using some tour construction heuristic or simply an arbitrary tour.

Step 2. Repeatedly attempt to improve the existing tour using local modifications.

2-Opt and 3-Opt Algorithms. The most frequently used local modifications are 2-swap and 3-swap. The 2-swap modification construct new tours by deleting two of the edges then reversing one of the two resulting paths and reconnecting the tour. In the 3-swap modification, the new tour is obtained by deleting three of the edges and reconnecting the three resulting paths in a new way, possibly reversing one or two of them. The 2-swap and 3-swap local modifications give rise to the most famous 2-Opt [Croes 1958] and 3-Opt algorithms [Lin 1956].

A straightforward way to implement the 2-Opt algorithm is to repeatedly consider all $(n\ (n-1)\ /\ 2)$ possible pairs of edges in the tour, until no improvement can be made. Therefore, each such operation takes $O\ (n^2)$ time in 2-Opt, and similarly, it takes $O\ (n^3)$ time to check all pairs of edges in the tour in 3-Opt algorithm. Unfortunately one can not predict the number of times to repeat this operation, thus one can not bound the running time of the overall algorithm. In practice, there are a lot of variety ways to implement the 2-Opt and 3-Opt algorithm. In this project, the 2-Opt and 3-Opt algorithms are implemented to run for n^2 number of iterations, using random choices of the links.

Other Variants of the 2-Opt and 3-Opt Algorithms. Some new algorithmic ideas have been developed for TSP. One that has drawn a lot of attention is Simulated Annealing [Kirkpatrick, 1983]. This algorithm can be viewed as a randomized variant on a 2-Opt or 3-Opt algorithm. The key difference of Simulated Annealing from the classic local optimization algorithm lies in two points:

1) Instead of examining all possible pairs of edges in the tour one by one, it chooses random pairs of edges.

2) This procedure sometimes allows modifications that make the tour worse, not necessarily improving the tour at each step.

The acceptability of modifications is controlled by a parameter called the temperature. The temperature t for the nth iteration is defined as $C/\log n$, where C is a fixed constant. Whether a new tour S' can be accepted is determined by the following principle. If the length of S' ($L(S')$) is less than the length of the original tour S ($L(S)$), then S' can be accepted; otherwise, let $\delta = L(S') - L(S)$, and r is a randomly chosen number from [0,1]; if $r \leq e^{-\delta/t}$, then S' is accepted. The basic structure of the Simulated Annealing algorithm is:

Step 1. Construct the initial tour $S = S_0$, and $t = t_0$.

Step 2. Choose a random pair of edges and reconstruct a new tour S'. If S' can be accepted, let $S = S'$. Then calculate the new t for the next iteration. Repeat Step 2 till it is time to quit.

Step 3. Return the best tour.

Another more recently developed algorithmic idea applied to TSP is the Genetic Algorithm. Brady was the first to propose the genetic idea for TSP [Brady 1985]. The basic idea for the genetic algorithm is:

Step 1. Start from k independent runs, each performing local optimization procedures. At the end of step 1, we have k locally optimized solutions.

Step 2. Derive *k* new starting solutions by transfer information among the *k* solutions found. This process is called mating. Repeat Step 1 and 2, until no progress can be made.

The mating strategy proposed by Brady [Brady 1985] is to mate two tours by finding a sub-path from each tour, so that the pair of sub-paths consist of exactly the same set of cities; then replace the longer one of the two paths by the shorter one. The advantage of a genetic algorithm is that it makes uses of the previous running results, while in the traditional local optimization algorithm, earlier results have no impact on later runs. However this genetic algorithm for TSP is quite time consuming, and the experimental result for a 64-city geometric instance is not better than running the 2-Opt many times independently within the same amount of time [Johnson 1990].

The Simulated Annealing and Genetic Algorithms are more valuable for theoretical research than application in solving TSP. The experimental data showed that they are significantly slower than the classic local optimization algorithm [Johnson and McGeoch 1997].

There are a lot active research going on to study Traveling Salesman Problem. Some interesting links are listed below for farther reference:

http://weber.u.washington.edu/~cvj/tsp/tspnew.html– Sensitivity Analysis for the Euclidean Traveling salesman Problem

http://mat.gsia.cmu.edu/GROUP94/0703.html– Algorihtms for Solving Traveling Salesman Problem.

http//www.astro.virginia.edu/~eww…th/TravelingsalesmanConstants.html-- Traveling Salesman Constants

2.2 World Wide Web and Java Language

The World Wide Web (WWW) is a relatively new technology, which is designed to enable global, interactive and distributed information systems to be constructed. The main interest of the WWW lies in its ease of access and its simplicity of setting up a web server. Java is a programming language specifically designed for writing programs on the Internet and/or Intranets. Java programs that are designed to run within a Web browser are called applets, while a stand-alone Java program is called a Java application.

Some special features make Java suitable to be used in writing web programs. Java contains built-in support for the common web problems. The executables for Java programs are extremely small, so that they can be loaded fast over relatively slow communication lines. Java constraints encourage security. A Java program cannot access anything on the client computer unless it is permitted specifically. Moreover, Java is platform-independent. The same program can be executed on a PC, a Mac, or even a UNIX machine, as long as there is a Java Virtual Machine (JVM) on that machine.

As a program language, Java is simple and powerful. Java is object-oriented, which enables a programmer to write modular programs that can be easily maintained. Its syntax is similar to C/C++, so that many C/C++ programmers are able to learn Java easily. Moreover, some features of C/C++ are removed on purpose, in order to keep the Java language simple and secure. One of the biggest changes is that Java eliminated the pointers in C/C++, which often cause memory leakage and are error-prone; instead Java employed a model of references and garbage collection.

CHAPTER III

DESIGN AND IMPLEMENTATION ISSUES

This project animates some approximation algorithms for Traveling Salesman Problem, using Java as the programming language and WWW as the medium. The design and implementation issues of this project are described in this chapter. In the design part, the system requirements, system functionality, user interface and main classes hierarchy are covered. The details of the strategies used in development are covered in the implementation part.

3.1 Design

3.1.1 System Requirements

Table 1 summarizes system requirements for the user side and the server side.

Server Side	• Unix system • NCSA HTTP Server
User Side	• Any platform with Java Virtual Machine (JVM) 1.0 or above • Or any platform with Internet Connection and with JVM 1.0 and HTML-compatible browser

Table 1. System Requirements

3.1.2 System Functionality

Limitations. The limitations of the Traveling Salesman Problem that can be solved by this program are:

1) Complete, symmetric TSP
2) Obey triangle equality
3) Up to 100 points, confined in a square plain.
4) The distance between two points is defined as the length of the straight line connecting the two points.

Algorithms Covered. Seven approximation algorithms for solving TSP are covered in this project, including five Tour Construction Heuristics (Table 2) and two Local Optimization Heuristics (Table 3).

Tour Construction Algorithms	**Specification**
Nearest Neighbor	The initial city is the first input point
Greedy	
Furthest Insertion	
Christofides	Using approximation algorithm calculate the minimum cost matching
Strip	

Table 2. Tour Construction Algorithms Covered in the project

Local Optimization Algorithms	**Specification**
2-Opt	• Pseudorandom choices of edges to delete • Fixed number of iterations
3-Opt	

Table 3. Local Optimization Algorithms Covered in the Project

3.1.3 Input and Output

The program enables users to input data (points) by double mouse clicks on the confined screen area. The user can select different algorithms, and the running results on the same set of data can be compared among different algorithms. This program graphically shows the path searching process of each algorithm. Users can select to show the search process step-by-step or run continuously until completed. The user interface is shown in Figure 1.

Double clicking in the central display area enable users to input data; a red point is displayed in the corresponding spot. The label *Cities* displays the number of points inputted by the user. There are seven radio boxes labeled with different algorithm names. Clicking one of them allows users to select different algorithms to solve the TSP. Repeatedly clicking the *Step* button, the tour searching process is shown step-by-step. In the meantime, the label *Tour Length* displays the current length of the tour. When completed, the *Step* button and the *Run* button are both disabled, and the *Complete* button turns to red. While clicking the *Run* button once, the final resulting tour is displayed, followed by the disabling of the *Step* and *Run* buttons, and changing color of the *Complete* button. If the user wants to try another algorithm to solve the same TSP, he can click the *Refresh* button, then the tour disappears, but the input data is still left on the screen. If the user wants to try a new set of data, he can click the *Clear* button; a blank center area is displayed.

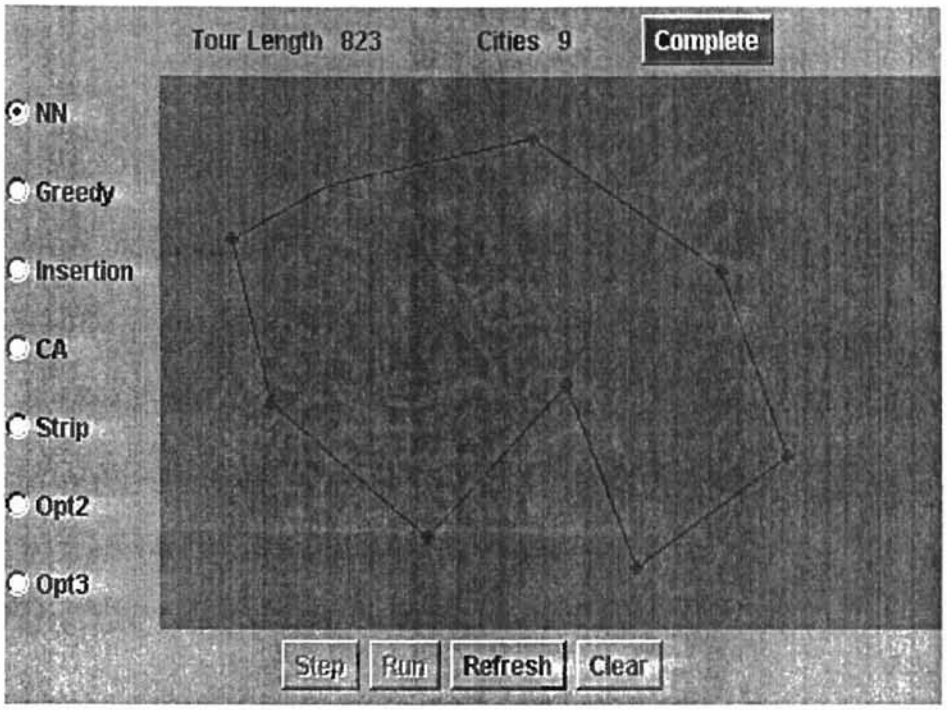


Figure 1. User Interface

If the 2-Opt or 3-Opt algorithm is selected, and when the user first clicks the *Step* button, another window is popped up. While the path searching process continues, a graph, which shows the relation of tour length and the number of iterations, is displayed in this new window (Figure 2). In this graph, the x-axis is the number of iterations and the y-axis is the tour length at each stage.

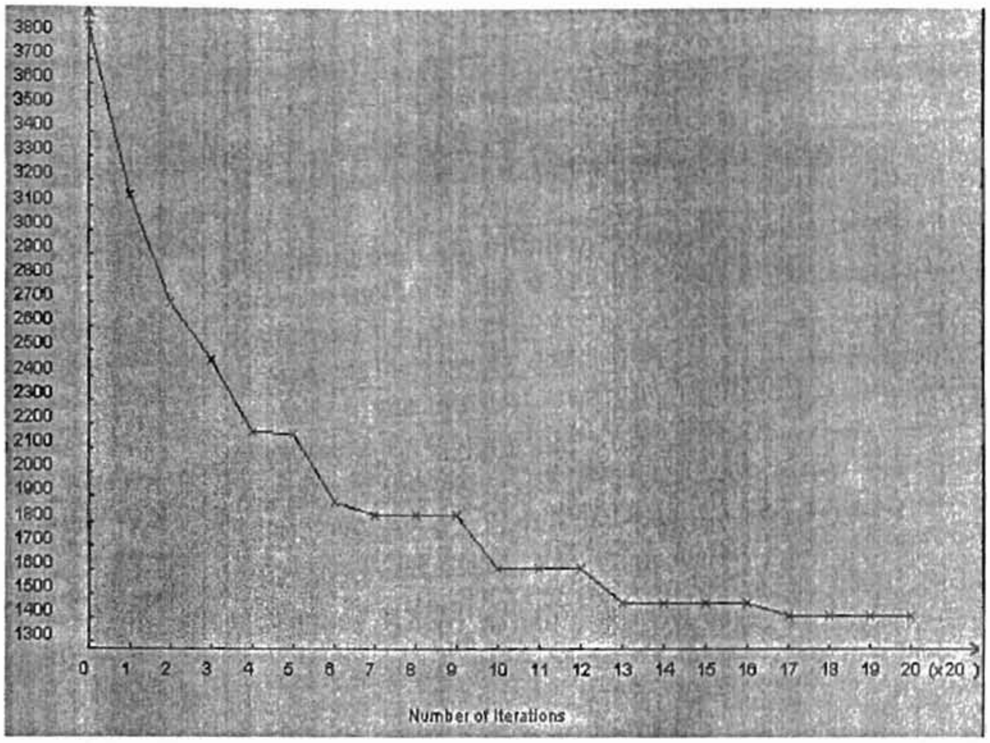


Figure 2. The Window Shows Tour Length vs. Iteration Number
For Local Optimization Algorithms

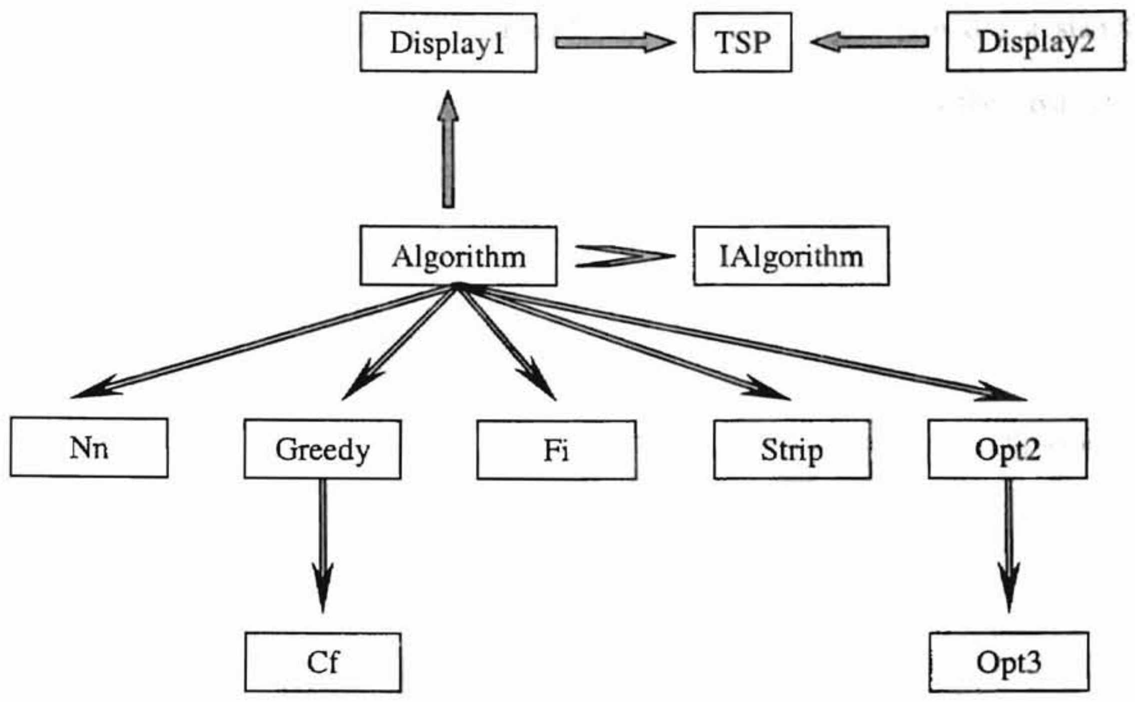
3.1.4 Hierarchy of main Classes

The main classes and a brief description of each class are summarized in Table 4. The detailed implementation is described in the next section.

TSP class is the main class for applet TSP. Instances of *Display1* and *Display2* are included in *TSP* class as data members. *Display1* includes an instance of *Algorithm* as data member, where *Algorithm* is a super class for all other algorithmic classes, i.e. *Nn*, *Greedy*, *Fi*, *Cf*, *Strip*, *Opt2* and *Opt3* are all inherited from *Algorithm* class. The hierarchy of the main classes is summarized in Figure 3.

Main Classes	**Description**
TSP	Implement the user interface, handle events
Display1	Implement the center display area
Display2	Implement the window that shows tour length vs. iteration number for local optimization algorithms
IAlgorithm	An interface, provide a common interface for all algorithms
Algorithm	Implement the IAlgorithm interface, providing some common methods shared among algorithms
Nn	Implement the Nearest Neighbor Algorithm
Greedy	Implement the Greedy Algorithm
Fi	Implement the Furthest Insertion Algorithm
Cf	Implement the Christofides Algorithm
Strip	Implement the Strip Algorithm
Opt2	Implement the 2-Opt Algorithm
Opt3	Implement the 3-Opt Algorithm

Table 4. Description of main Classes



⇨ : denotes a relation of is a member of, A ⇨ B means A is a member of B.
≫ : denotes a relation of implement, A ≫ B means A implement B.
⟹ : denotes a relation of inheritant A ⟹ B means B is inheritant from A.

Figure 3. Hierarchy of Main Classes

3.2 Implementation

This project implements seven approximation algorithms for the Traveling Salesman Problem with a user-friendly interface and using the World Wide Web as medium. Some implementation details of this project are discussed in this section.

3.2.1 Implementation of the User Interface

TSP Class. The *TSP* class is the Main class for applet TSP. It implements the user interface, and also contains the methods that control the life cycle of applet TSP. The

TSP class is inhereted from the *java.applet.Applet* class, which provides a standard interface between applets and their environment, such as a Web page or the Java Applet Viewer. The data members in TSP class are summaried in Table 5, and its methods are summaried in Table 6.

Data Member	**Description**
private boolean m_fStandAlone = false;	m_fStandAlone is set to true if the applet is run as a standalone application
private Point[] cities;	An array of Point, store the values of x cordinate and y cordinate of each city point. Each Point is identified by the index in the array, which is also in the order of user input, ie. *cities[0]* is the first input point, *cities[1]* is the second,...*cities[n]* is the *n+1th.*
private int num_cities;	The number of cities
private boolean bStart = true;	It is set to false after execution of the first step of each algorithm
private Panel top, bottom, left;	• Panel top: contains the TourLength and lcityNum Labels, and btComplete Button. • Panel bottom: conatins the btStep, btRun, btRefresh, btClear Buttons • Panel left: contains the CheckboxGroup chGroup
private Button btStep, btRun, btRefresh, btClear, btComplete;	Declarations of Buttons
private Label TourLength, lcityNum;	Declarations of Labels
private CheckboxGroup chGroup;	Contains the following Checkboxes
private Checkbox chNN, chGreedy,chInsertion,chCA, chStrip, chOpt2,chOpt3;	Declarations of Checkboxes
private Display1 disp1;	An instance of class *Display1*, which implements the displaying of each algorithm in the central dispay area
private Display2 disp2;	An instance of class *Display2*, which implements the window showing the relation of tour length vs. iteration number for local optimization algorithms

Table 5. Data Member in *TSP* class

Methods	**Description**
public static void main(String args[])	Acts as the applets entry point when it is run as a standalone application. It is ignored if the applet is run from within an HTML page.
public TSP()	TSP Class Constructor
public String getAppletInfo()	Returns a string describing the applets author, copyright date, or miscellaneous information
public void init()	Called by the AWT when an applet is first loaded or reloaded, to define the user interface.
public boolean action(Event evt, Object arg)	Called when Buttons is clicked, and handle the events
public void actStart()	Called by *action()*, when *Step* button is clicked for the first time in each algorithm, to create the *Algorithm* instance for *Display1*.
private void IS_compl(boolean f)	Called by the *action()*, to change the color of *Complete* button to red or gray.
public boolean handleEvent(Event evt)	Called when the close window event occurrs for the standalone application

Table 6. Methods in *TSP* class

The *main()* method is a standalone application support. The function includes: creates Top-level Window to contain applet TSP and shows the Window Frame; starts the applet running within the frame window and sets *m_fStandAlone* to true to prevent *init()* from trying to get them from the HTML page.

The *action()* method is called automatically, when a button is clicked. Different actions are taken when different buttons are clicked. The *action()* method is described in pseudo-code as follows:

```
public boolean action(Event evt, Object arg)
{
  if Step button is clicked, then
        if it is the first step then
             call actStart() to create the Algorithm instance in Diasplay1;

        call step() method in Display1, to display the new tour;
        display current tour length;
        change the color of Complete button if necessary;

        if 2-Opt or 3Opt is chosen, then
             call setPoint() in Dispaly2 to add a new point to the graph;
             call drawPoint() in Display2 to draw the graph;

        if last step completed, then
             disable Step and Run button;

   else if Run button is clicked, then
        call actStart() to create the Algorithm instance in Diasplay1;
        call run() method in Display1, to display final tour;
        display final tour length;
        change the Complete button to red
        disable Step and Run button;

   else if Refresh button is clicked, then
        call refresh() method in Display1, to clear the tour;
        change the color of Complete button to gray;
        enable Step and Run button;
        Dispose the instance of Display2 if there is one

   else if Clear button is clicked, then
        call the clear() method in Dispaly1, to display a blank screen;
        change the color of Complete button to gray;
        enable Step and Run button;
        Dispose the instance of Display2 if there is one
 }
```

The *Display1* Class. The class *Display1* is responsible for getting the input cities from the user, constructing the tour using the user-selected algorithm and drawing the tour at each stage in the center display area. *Display1* extends the *java.awt.Canvas* class. A *Canvas* component represents a blank rectangular area of the screen, where the application can draw objects or trap input events from the user. The data members and methods of the *Display1* class are summarized in Table 7 and Table 8.

Data Member	**Description**
private Point cities[];	An array of Points stores the user-input data.
private int cityNum;	Number of Point inputted
private Graphics G;	The graph to paint on
public Algorithm alg;	The instance of *Algorithm* class, initiated in the *actStart()* method in *TSP* class
private int stepNum;	Define the number of *Step()* to be called in *Run()* method.
private Label lCN;	A reference to the *lcityNum* of *TSP* class

Table 7. Data Members in *Display1* class

The *Display2* class. The class *Display2* is responsible for displaying the window which shows the relation of tour length and the number of iterations for the 2-Opt and 3-Opt algorithms. It extends the *java.awt.Frame* class, which is a top-level window with a title and a border. The instance of *Display2* in *TSP* class is constructed in *actStart()* method when 2-Opt or 3-Opt is chosen to solve the TSP. The data members and the methods in the *Display2* class are summarized in Table 9 and Table 10.

Method	**Description**
Display1(Label l)	The constructor of *Display1* class; the parameter is the *lcityNum* of *TSP* class. Called by the *init()* method of *TSP* class. It initiates the *Point* array *cities*, sets *cityNum* to zero, *lCN* reference to the *l*,
public void setStepNum(int n)	Called by the *actStart()* method in *TSP* class to set the *stepNum*.
public void paint(Graphics g)	Called whenever the *repaint()* is called or when Window is moved or resized in the standalone application. Responsible for drawing the input points, calls the *drawTour()* routine of the *Algorithm* to draw the tour.
public boolean step()	Called by *action()* method in *TSP* class, when the *Step* Button is clicked. Calls the *step()* method in the corresponding *algorithm*, and *repaint()*. Returns true if last step is completed, otherwise returns false.
public void run()	Called by *action()* method in *TSP* class, when the *Run* Button is clicked. Calls the *run()* method in the corresponding *algorithm*, and *repaint()*.
public void refresh()	Called by *action()* method in *TSP* class, when the *Refresh* Button is clicked to erase the tour.
public void clear()	Called by *action()* method in *TSP* class, when the *Clear* Button is clicked to clear the screen.
public boolean mouseDown(Event evt, int x, int y)	Called whenever the mouse is clicked down. Records the current mouse position by creating a new Point, and stores it in the Point array *cities*.
public Point[] getCities()	Called by the *actStart()* method in *TSP* class. Return the *cities* to the calling routine
public int getNum()	Called by the *actStart()* method in *TSP* class. Return the *cityNum* to the calling routine

Table 8. Methods in *Display1* class

Data Member	Description
private int cityNum;	Number of cities
private Point[] p;	A Point array; in each Point, x coordinate represents the iteration number, y coordinate represents the tour length
private DrawCanvas c;	An instance of *DrawCanvas* class, which takes care of the drawing of graph
private int ind;	The number of Points in the graph

Table 9. Data Members of *Display2* class

Display2() is the constructor for *Display2* class. It takes the title of the window and the number of cities as parameters, initializes *ind* to zero, and creates the instance of *Drawcanvas* class. *DrawPoint()* method gets the dimension of the current window, and calls the *draw()* method in *DrawCanvas* class, passing the Point array *p*, dimension information and *ind* as parameters.

Method	Description
Display2 (String title, int n)	Constructor, called in *actStart()* method in *TSP* class when 2-Opt or 3-Opt is chosen to solve the TSP
public void setPoint(int l)	Called after *step()* is called in the *action()* method in the TSP class to record the tour length and the iteration number, if 2-Opt or 3-Opt is chosen to solve the TSP
public void drawPoint()	Called after every time *setpoint()* is called
public boolean handleEvent(Event evt)	Called whenever a close window event occurred

Table 10. Methods in *Display2* Class

The *DrawCanvas* class handles the actual drawing of graph. Its *draw()* routine first transforms each Point in the Point array *p* to a corresponding Point, whose x and y coordinates represent the actual position in the window frame, then calls ***repaint()***, which in turn calls the *paint()* method to paint the graph.

3.2.2 Implementation of Approximation Algorithms

The *IAlgorithm* Interface. *IAlgorithm* is an interface for all classes that implement algorithms. An interface contains members that are constants and abstract methods. It has no implementation, but other classes can implement it by providing implementations for its abstract methods. The abstract methods in *IAlgorithm* are summarized in Table 11.

Abstract Method	**Description**
public boolean step();	Implements the steps to search the path; returns true when complete, otherwise returns false
public int getLength();	Returns the current tour length to the caller routine

Table 11. The Abstract Methods in *IAlgorithm* Interface

The *Algorithm* class. The *Algorithm* class implements the *IAlgorithm* interface and it is a super class for all classes implementing the individual algorithm. The data members and methods in the *Algorithm* class are summarized in Table 12 and Table 13.

You may have noticed that all data members in the *Algorithm* class are *protected* instead of *private*. The *protected* keyword specifies that those members can be accessed by its derived classes, in addition to its member functions.

Data Member	Description
protected int visit;	The number of cities visited
protected int tour[] = new int[101];	Specifies the order of the tour
protected int vb[] = new int[100];	If a city is visited, set the corresponding element in *vb* array to *1*, otherwise set to *0*
protected int cityNum;	Number of cities
protected Point cities[];	An array of Point to store the position of each city
protected int matrix[][] = new int[100][100];	The distance matrix between each pair of cities

Table 12. The Data Members in *Algorithm* class

Method	Description
Algorithm(Point[] A, int n)	Constructor, called by every constructor of the derived class
public void consMatrix()	Called by *Algorithm()* to construct the distance matrix.
public void init()	Called by *step()* method, to construct the initial tour.
public void drawTour(Graphics g)	Called by the *paint()* method in *Display1*, to draw the tour according to the order specified by the integer array *tour*.
public boolean step()	Called by the *step()* method in *Display1*, to actually carry out the construction of the tour
public int next(int i)	Called by the *step()* method to determine the next city to visit
public void add2tour(int i)	Call by *step()* method to add the next city to visit to the exiting tour
public int getLength()	Called by *step()* method in *Display1* to get the current tour length

Table 13. The Methods in *Algorithm* Class

The *step(), DrawTour()* and *getLength()* methods are shared among some of the derived classes of *Algorithm*. While *init(), next()* and *add2tour()* need to be over-written by the derived classes. The *step()* method is described in pseudo-code as follows:

```
public boolean step()
{
    if  it is the first step, then
            call init() to construct the initial tour;
            return flase;
    else if not all the cities have been visited, then
            call next() to determine the next city to visit, or derive a new tour for Opt2 and Opt3;
            call add2tour() to add the new city to the tour;
            return false;
    else close tour by returning to the initial city; return true;
}
```

The main purpose of the *Algorithm* class is to provide a common implementation for all the derived classes. All the classes that implement TSP algorithms in this project have the same basic structure as the *Algorithm* class. Therefore, in the following section where the classes derived from the *Algorithm* class are discussed, the main focus is on how the *next()* , *init()* and *add2Tour()* methods are implemented differently.

The *Nn* Class. The *Nn* class is the class that implements the Nearest Neighbor algorithm. The *init()* method simply constructs the initial tour as the first city that the user inputted. The code for *init()* is as follows:

```
public void init()
{
    tour[0] = 0;  // choose the first input city as the initial city
    vb[0] = 1;    // set the vb of this city to 1;
    visit ++;     // increment the number of cities visited
}
```

The *next()* method takes the current city as the parameter, and returns the closest city to the current one as the next city to visit. The pseudo-code for *next()* is as follows:

```
public int next(int n)
{
    set d to a number that is longer than any edge
    loop for j = 0 to cityNum, j++
        if city j is unvisited, and distance between n and j is less than d, then
            update d as this distance;
            set the next city as j;
    end loop
    return next city;
}
```

The *add2tour()* for the Nearest Neighbor algorithm is very simple; just add the next city to visit at the end of the tour. The code for *add2tour()* is given below.

```
public void add2tour(int n)
{
    tour[visit] = n;
}
```

The *Greedy* Class. *Greedy* is the class to implement the Greedy algorithm. The *tour* in *Greedy* class is represented by an array of *Edge*, where the *Edge* is a class whose data members and methods are summarized in Table 14 and 15. In the constructor of the *Greedy* class, a priority queue of all the Edges is built based on the length of the Edges.

Data Member	**Description**
private int start;	The start vertex of the edge
private int end;	The end vertex of the edge
private int distance;	The length of the edge

Table 14. The Data Member of *Edge* class

Method	Description
Edge(int s, int e, int dis)	The constructor
public int getStart()	Returns the start vertex to the calling routine
public int getEnd()	Returns the end vertex to the calling routine
public int getDistance()	Returns the length of the edge to the calling routine

Table 15. The Methods in the *Edge* Class

The *init()* method generates the initial tour from the minimum length edge. The pseudo-code of *init()* is shown below.

```
public void init()
{
   call the delMin() of the priority queue to get the min_Edge
   add this min_Edge to the tour
}
```

The *next()* method is to find the shortest feasible Edge, which is described in pseudo-code as follows:

```
public Edge next()
{
     loop until next Edge is found
          Call delMin() to get the minimum remaining Edge
          If it is feasible then
              Return the Edge
     end loop
}
```

The feasibility of an Edge is checked by the *feasible()* routine; the pseudo-code of it is shown as follows:

```
private boolean feasible(Edge eg)
{
    if the two ends of eg belong to the same set, then
        return false;
    else if the degree of the start edge >= 2 or the degree of the end edge >= 2, then
        return false;
    else
        Union the two vertex of eg to one set;
        Return true;
}
```

The *Union/Find* Algorithm [Weiss 1997] is used to determine if the two vertices belong to the same set. Two vertices belong to the same set if and only if they are connected in the current tour. Adding an edge with two ends in the same set will create a cycle in the tour.

The *Fi* Class. The *Fi* class implements the Farthest Insertion algorithm. The key point in implementation of this algorithm is to keep an array *minEdge* to record the distance of each city to its nearest neighbor that is in the current tour. For example, if city *i* itself is in the tour, then *minEdge*[*i*] = 0; else if the current tour consist *a*, *b*, *c*, *d* , and the distance between *i* and the four cities in tour is:

	a	b	c	d
i	10	35	20	98

then minEdge[*i*] = 10.

The *init()* method is to find the two cities with the longest distance, and include the two cities in the tour. The pseudocode for *init()* is listed below:

```
public init()
{
    find the 2 cities i and j, which have the longest intercity distance;
    add i and j to the tour;
    construct the minEdge array;
}
```

The next city to be included in the tour is simply the city that has the maximum value in the *minEdge* array. The pseudo-code for the *next()* method is shown as follows:

```
public int next()
{
    find the index of the max value in the minEdge array
    return the index as next city to visit;
}
```

The *add2tour()* routine is to add the next city to the tour so that the increase of the total tour length is minimized. The pseudo-code for add2tour() is shown below:

```
public add2tour(n)
{
    int min = 10000;
    loop for i= 0 to visit, i++
        increase = distance(tour[i], n) + distance(n, tour[i+1]) – distance (tour[i],tour[i+1]);
        if increase < min, then
            insert_position = i+1;
    end loop
    loop for j = visit to insert_position; j--
        tour[j] = tour[j-1];
    end loop
    tour[insert_position] = n;
}
```

The *Cf* Class. The Cf class implements the Christofides Algorithm. It extends the *Greedy* class, because Kruskal's algorithm to construct the minimum spanning tree is quite similar to the Greedy algorithm to construct the TSP tour. The only difference lies in the checking of the feasibility of the edge; in the Kruskal's algorithm, an edge is feasible as long as the two ends of the edge do not belong to the same set, no matter if the degree of any vertex exceeds two.

The *step()* method in *Cf* class is different from that of the *Algorithm* class. The pseudocode is given below:

```
Public boolean step()
{
    if it is the first step, then
        call init() to construct the minimum spanning tree;
    else if it is the second step, then
        call the minMatch() to computer the minimum cost matching of the odd degree vertex in the
        tree;
    else
         call traverse(), to construct the tour by traversing the graph

    if tour is completed, then
         return true;
    else return false;
}
```

The *init()* method is to construct the minimum spanning tree, which is quite similar to constructing the greedy tour. *minMatch()* computes the minimum cost matching of the odd degree vertex in the MST. The Greedy heuristic for the matching is used in this project. The heuristic repeatedly finds the two closest points in the set, if the two points are both unmarked, then it adds the edge defined by the two points to the matching edges and marks the two points; it quit when all the points are marked. The pseudo-code for *minMatch()* is shown as follows:

```
private void minMatch()
{
        Edge matchHeap[];
        int matchList[];

        loop for j = 0 to cityNum, j++
            If city j has odd degree in MST, then
                Add j to matchList;
        end loop

        construct the priority queue matchHeap from the matchList, according to the length of the
        edge ;

        loop while ind < matchList.size/2
            call delMin(matchHeap), to get the shortest edge;
            if both ends of the edge are not included in the matching edges, then
                add the edge to the matching edges;
                ind++;
        end loop
}
```

The *traverse()* routine constructs the tour by traversing the graph containing the MST and the matching edges. The pseudo-code is shown as below:

```
private void traverse()
{
      int current = 0;
      int next;
      tour[0] = 0;
      loop while not all the cities are in the tour
          next = neighbor(curent);
          current = next;
          if next is unvisited, then
              include next to tour;
      end loop
}
```

The neighbor of the current city is a city that is connected to the current one in the graph. The *neighbor()* method returns the nearest unvisited neighbor of the current city, or if all the neighbors to the current city are visited, returns one of the neighbors to the current city, which has unvisited neighbors.

The *Strip* Class. The *Strip* class is to implement the Strip algorithm. The *Step()* methods is also different from that of the *Algorithm* class.

```
Public boolean step()
{
    if it is the first step, then
        call slice(), to slice the unit square into $\sqrt{n}$ vertical equal-width strips;
        call partition(), to partition the cities to different strips according to the x coordinates;
        call sortY(), to sort the cities in each strip according to the y coordinates;
    else
        call traverse() to construct the tour by traverse the cities in the strips;

    if tour is completed, then
        return true;
    else return false;
}
```

The implementations of *slice()*, *partition()* and *sortY()* are straight forward. The tour is constructed in the *traverse()* routine, whose pseudo-code is given below:

```
private  void traverse()
{
    loop for $j = 0$ to No_of_stript , $j$++
        if $j$ is even, then
            call up() to go through the cities in the strip from bottom to top, and add each city
            encountered to the end of the tour;
        else
            call down() to go through the cities in the strip from top to bottom, and add each
            city encountered to the end of the tour;
        end loop
    }
```

The *Opt2* Class. The Opt2 class implements the 2-Opt algorithm. The *init()* method generates an initial tour, which is in the same order as user input. The *next()* method generates a new tour by *cityNum* invocations of the 2-swap operation, whose pseudo-code is given below:

```
private void next()
{
    loop for i = 0 to cityNum, i++
        Randomly choose 2 edges;
        Get 4 vertices from the 2 edges, ...v1→ v2→.. abc.. →v3→v4...;
        diff = distance(v1,v3) +distance(v2,v4) –distance(v1,v2) – distance(v3,v4);
        if (diff <0)
            reverse the path between v2 and v3 to get the new tour
            (...v1→ v3→.. cba.. →v2→v4...)
    end loop
}
```

The *Opt3* Class. The *Opt3* class implements the 3-Opt algorithm. It is obtained by extending the *Opt2* class with revised *next()* method. The pseudo-code of *next()* method in Opt3 class is shown in below:

```
private void next()
{
    loop for i = 0 to cityNum, i++
        Randomly choose 3 edges;
        Get 6 vertices from the 3 edges, ...v1→ v2→.. abc.. →v3→v4→..def..→v5→v6...;
        diff =  matrix[v1][v4] + matrix[v2][v5] +matrix[v3][v6]-
                matrix[v1][v2] - matrix[v3][v4]-matrix[v5][v6];

        if (diff <0), then
```

```
            reverse the path between v2 and v3;
            reverse the path between v4 and v5;
            reverse the path between v1 and v6;
        else
            diff = matrix[v1][v4] + matrix[v2][v6] + matrix[v3][v5] -
                   matrix[v1][v2] - matrix[v3][v4]-matrix[v5][v6];
            if (diff < 0)
                reverse the path between v4 and v5;
                reverse the path between v1 and v6;
    end loop
}
```

CHAPTER IV

RESULTS

The program can be executed successfully with the most popular Internet browsers: Netscape Navigator and Internet Explorer. The examples of the user interface in Netscape and Internet Explorer are shown in Figure 4 and 5. The program also can be run as a standalone application. A sample snapshot of a standalone application is shown in Figure 6.

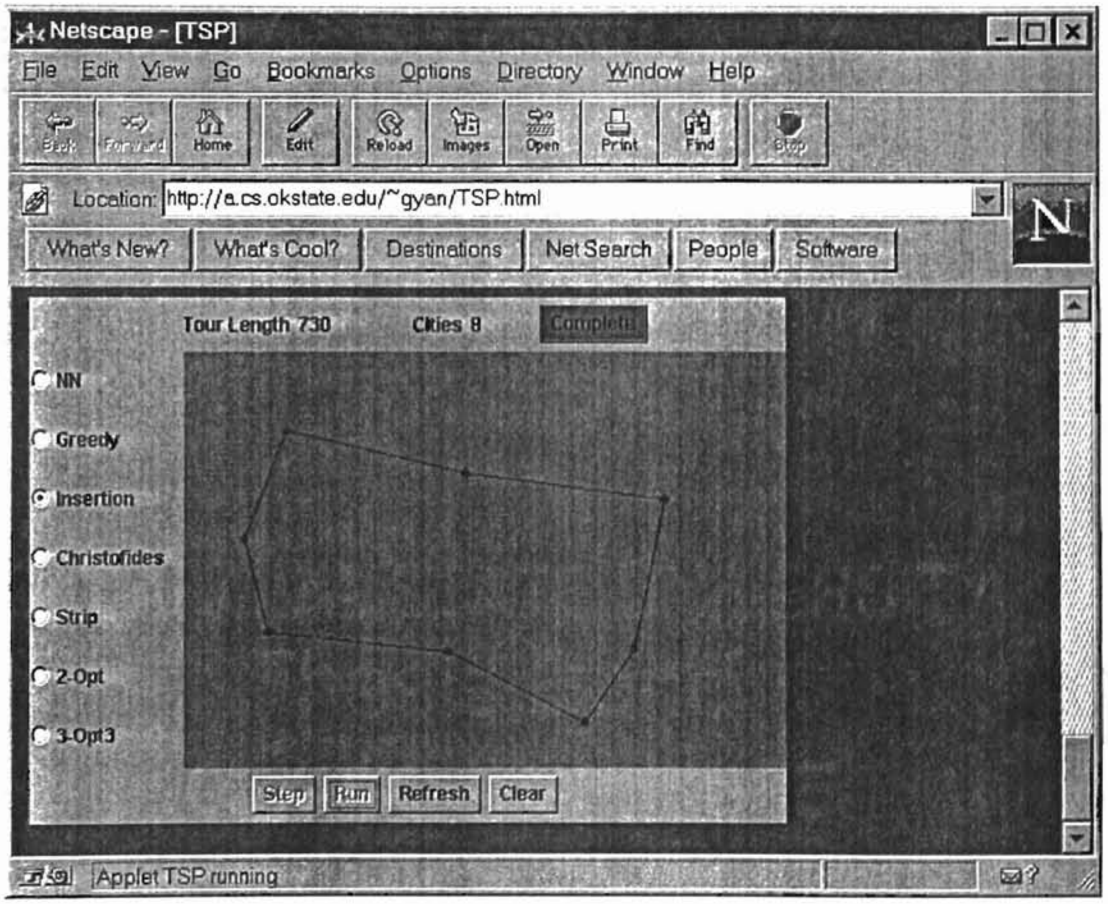

Netscape - [TSP]
File Edit View Go Bookmarks Options Directory Window Help
Back Forward Home Edit Reload Images Open Print Find Stop
Location: http://a.cs.okstate.edu/~gyan/TSP.html
What's New? What's Cool? Destinations Net Search People Software
Tour Length 730 Cities 8 Complete
NN
Greedy
Insertion
Christofides
Strip
2-Opt
3-Opt3
Step Run Refresh Clear
Applet TSP running

Figure 4. A Sample Screen in Netscape

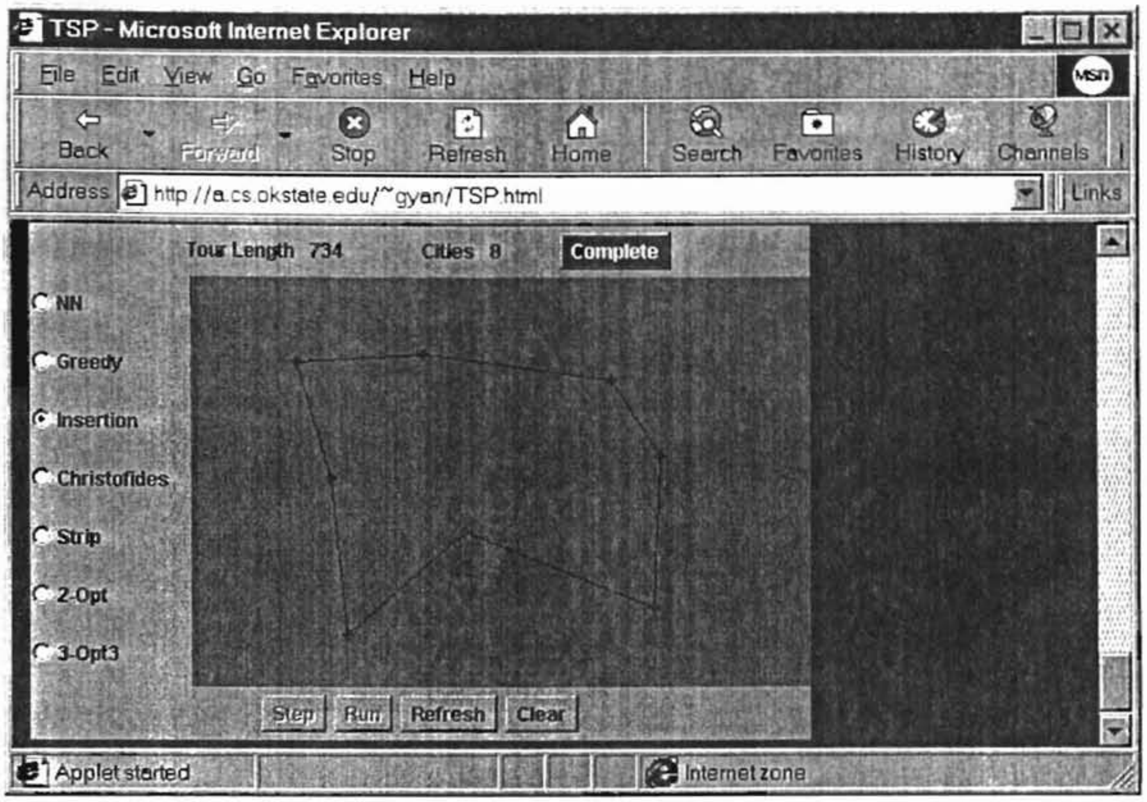

TSP - Microsoft Internet Explorer
File Edit View Go Favorites Help
Back Forward Stop Refresh Home Search Favorites History Channels
Address http://a.cs.okstate.edu/~gyan/TSP.html
Links
Tour Length 734
Cities 8
Complete
NN
Greedy
Insertion
Christofides
Strip
2-Opt
3-Opt3
Step Run Refresh Clear
Applet started
Internet zone

Figure 5. A Sample Screen in Internet Explorer

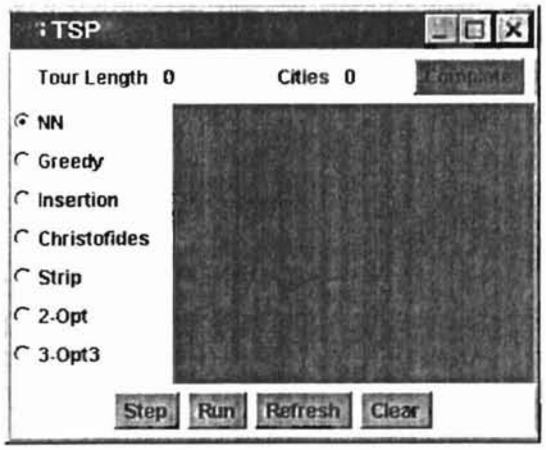


Figure 6. A Sample Snapshot of a Standalone Application

The initial screen is shown in Figure 7. The default algorithm is the Nearest Neighbor algorithm. The user can input data by double clicking in the central area, and choose the algorithm by clicking on the name of the algorithm. The sample screen is shown in Figure 8. The sample snapshots during the execution of each algorithm are shown in Figure 9 to Figure 15.

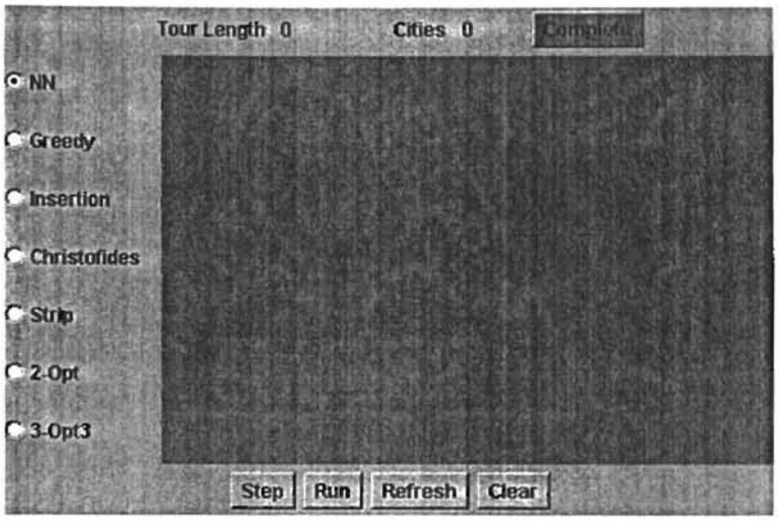


Figure 7. The Initial Screen

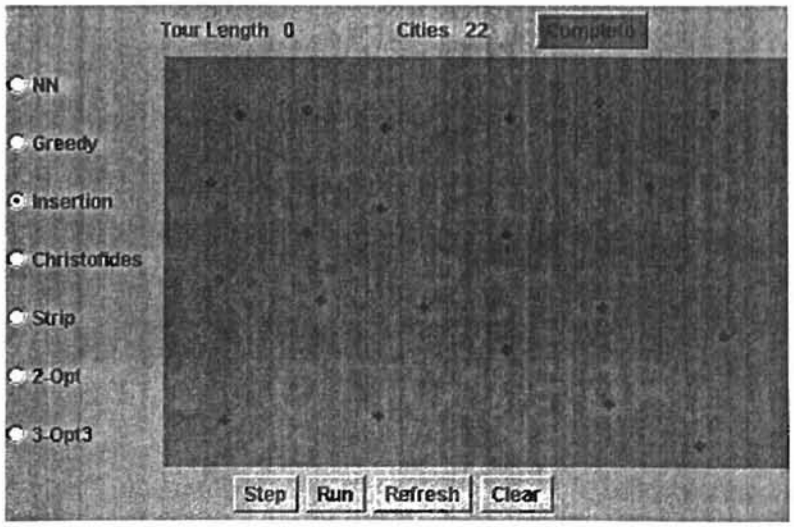
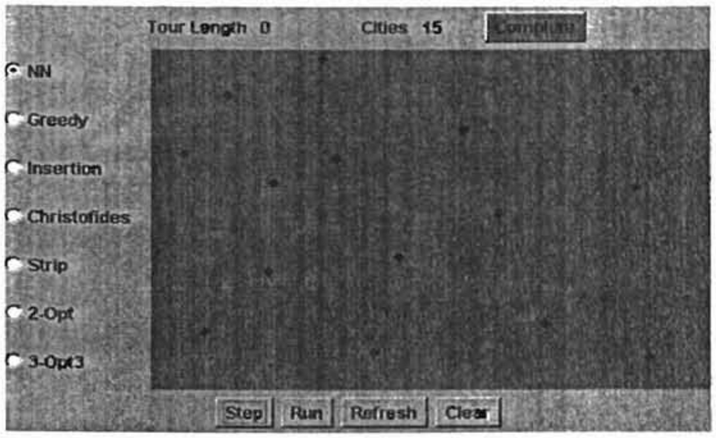
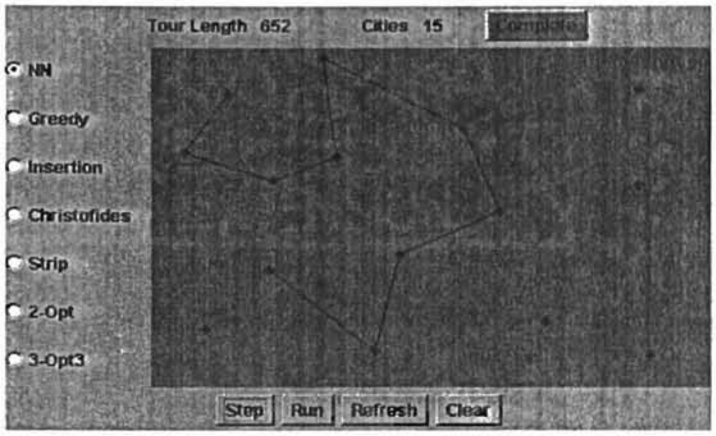


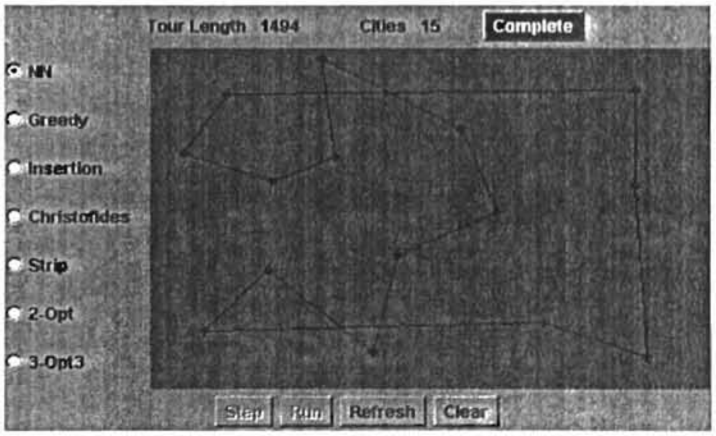
Figure 8. A Snapshot of Getting User Input



(a)

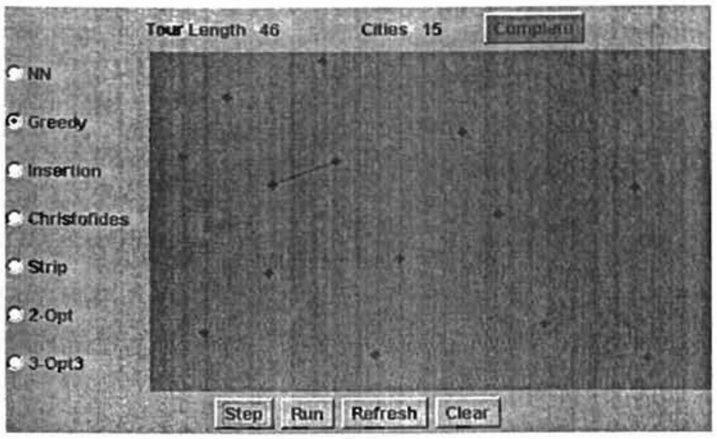


(b)

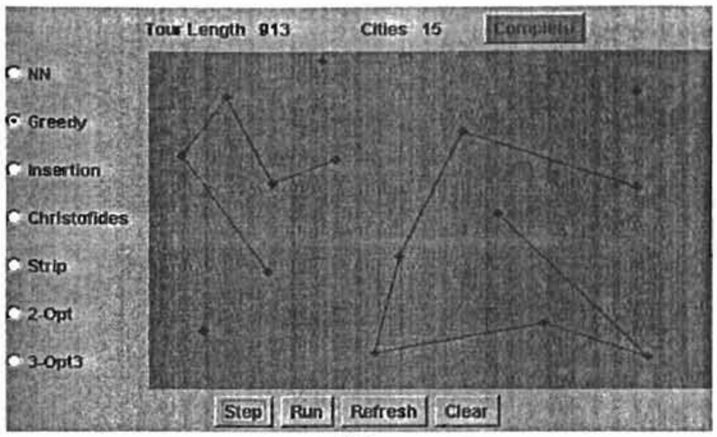


(c)

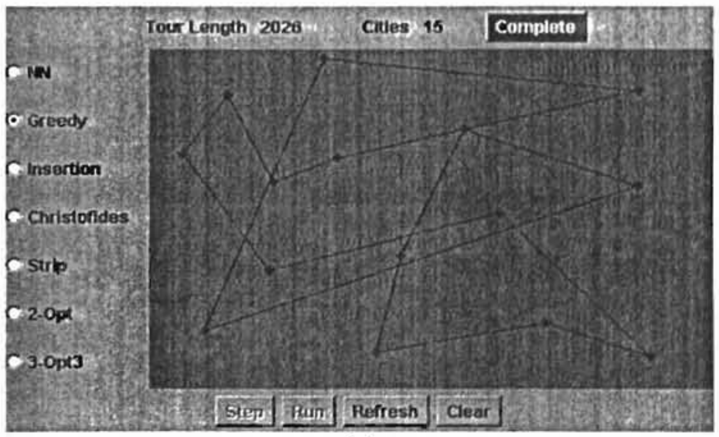
Figure 9. Sample Snapshots of Running the Nearest Neighbor Algorithm
(a) after first step, (b) after10 steps, (c) completed



(a)

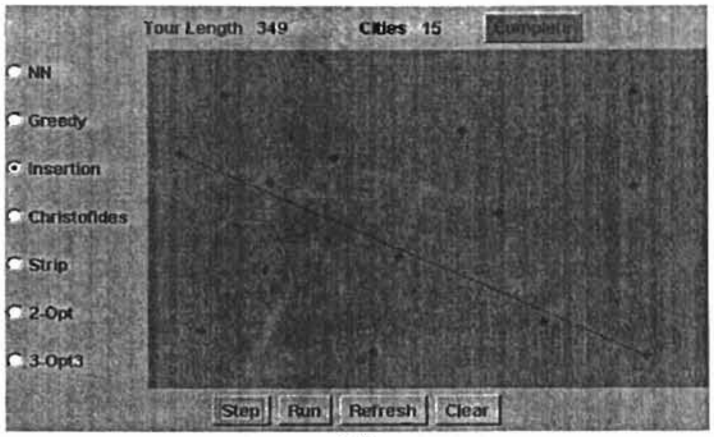


(b)

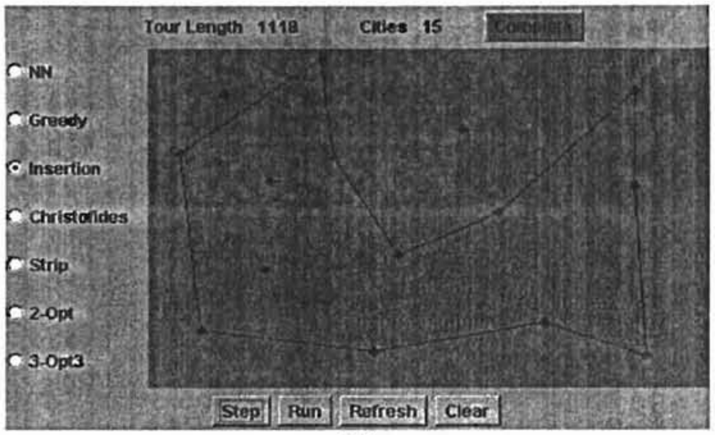


(c)

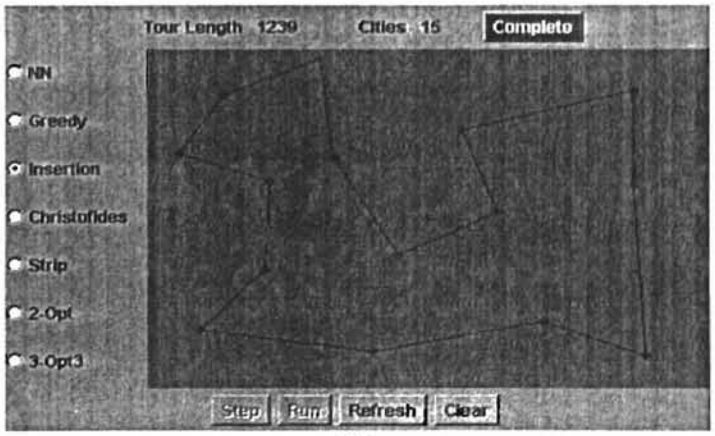
Figure 10. Sample Snapshots of Running the Greedy Algorithm.
(a) after first step, (b) after 10 steps, (c) completed



(a)

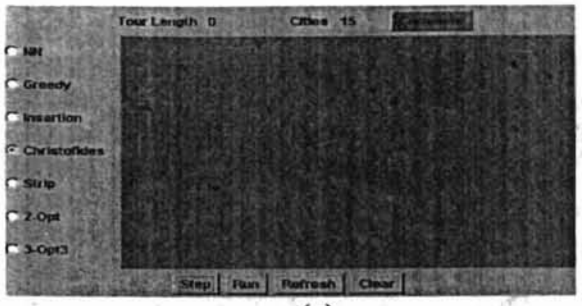


(b)



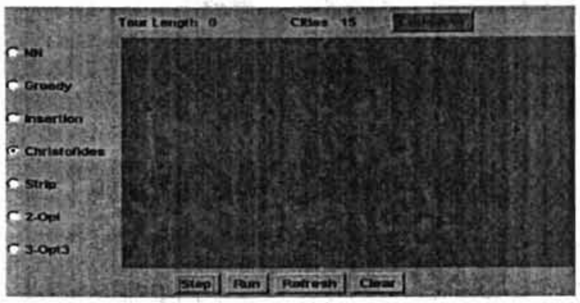
(c)

Figure 11. Sample Snapshots of Running the Farthest Insertion Algorithm
(a) after first step, (b) after 10 steps, (c) completed


Tour Length 0 Cities 15

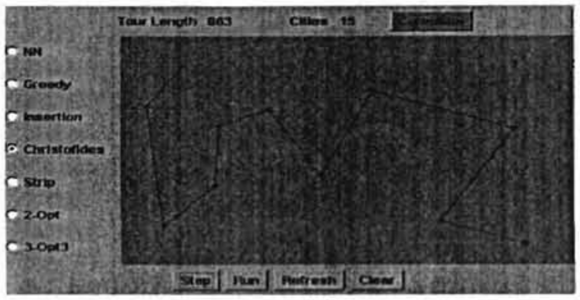
[✓ Christofides]

(a)


Tour Length 0 Cities 15

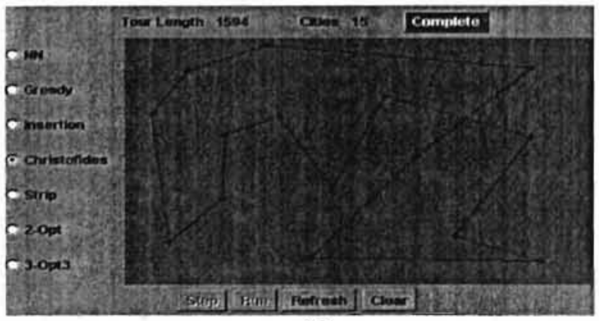
[✓ Christofides]

(b)


Tour Length 863 Cities 15

[✓ Christofides]

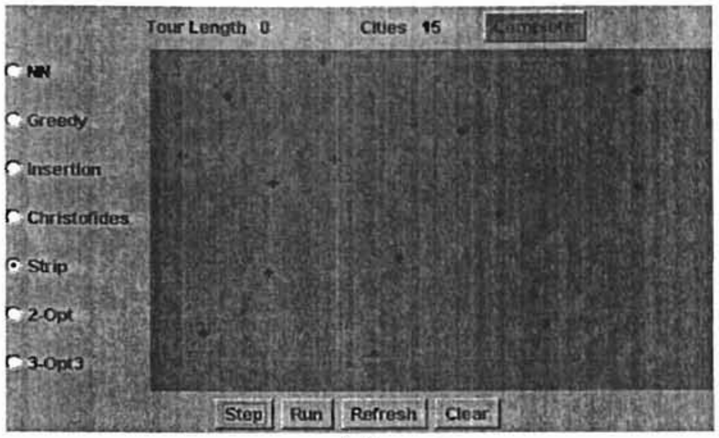
(c)


Tour Length 1594 Cities 15 Complete

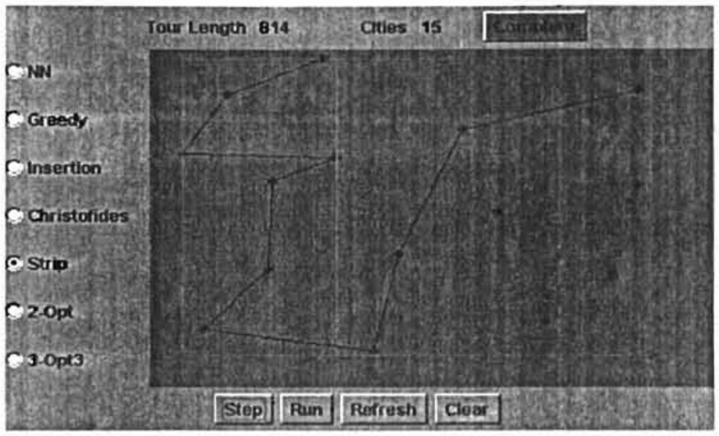
[✓ Christofides]

(d)

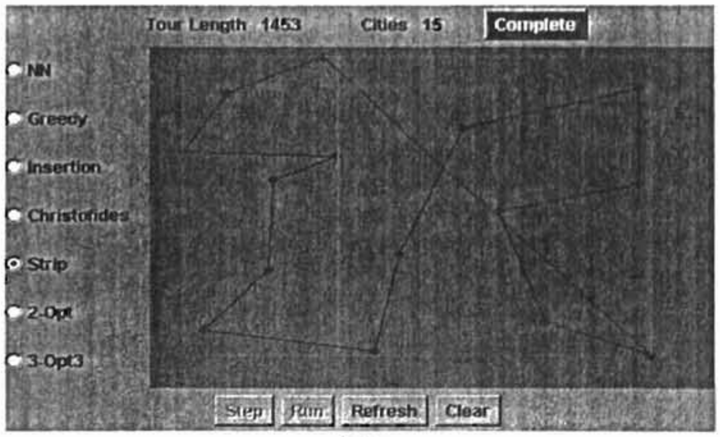
Figure 12. Sample Snapshots of Running the Christofides Algorithm
(a) Minimum Spanning Tree (MST), (b) MST plus minimum cost matching edges
(b) traversing the graph, (d) completed



(a)

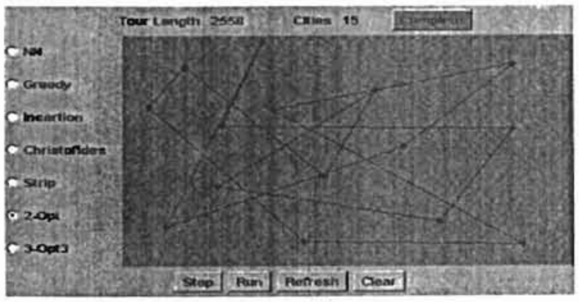


(b)

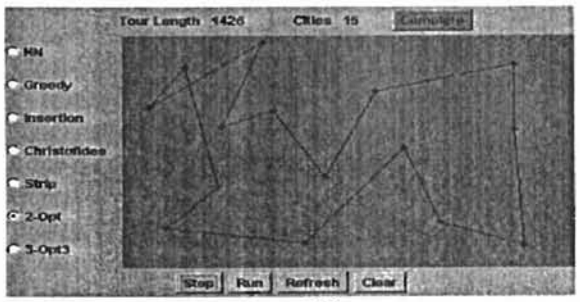


(c)

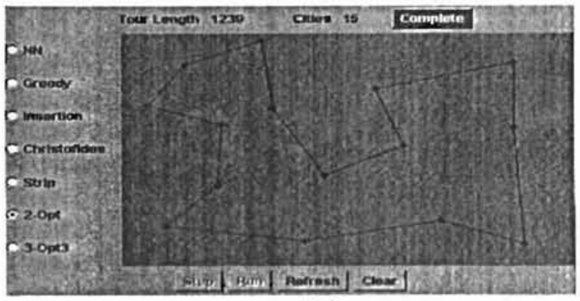
Figure 13. Sample Snapshots of Running the Strip Algorithm
(a) divide the unit into strips, (b) traversing the cities in strips, (c) completed



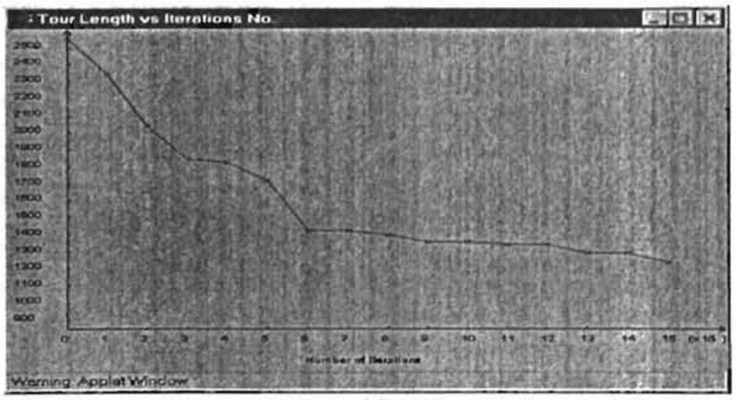
(a)



(b)

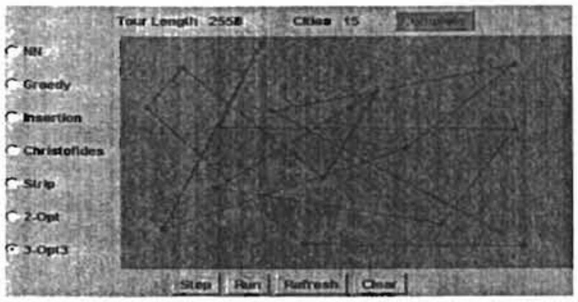


(c)

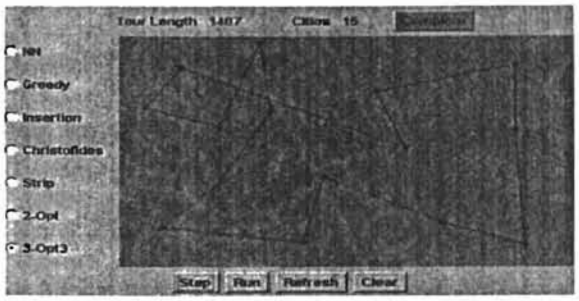


(d)

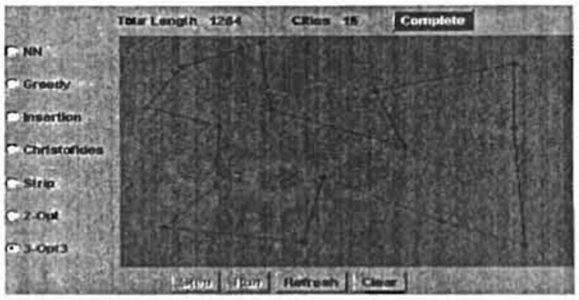
Figure 14. Sample Snapshots of Running the 2-Opt Algorithm
(a) initial tour, (b) after seven steps of improvement, (c) completed,
(d) the window showing the relation of tour length and iteration number


Tour Length 2558 Cities 15

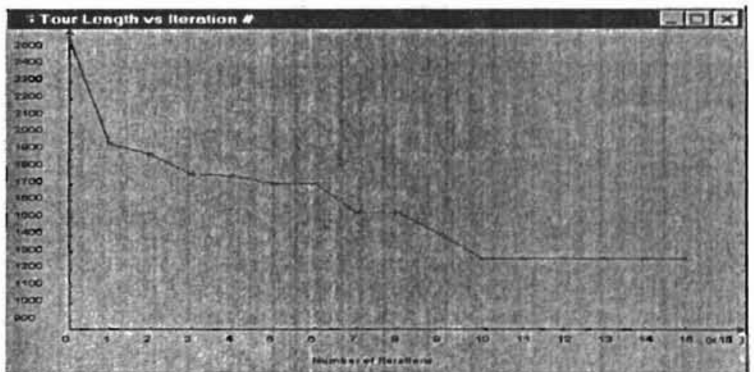
(a)


Tour Length 1487 Cities 15

(b)


Tour Length 1284 Cities 15 Complete

(c)



(d)

Figure 15. Sample Snapshots of Running the 3-Opt Algorithm
(a) initial tour, (b) after nine steps of improvement, (c) completed,
(c) the window showing the relation of tour length and iteration number

CHAPTER V

SUMMARY AND FUTURE WORK

5.1 Summary

This project animates seven approximation algorithms for the Travelling Salesman Problem, including five Tour Construction Heuristics – Nearest Neighbor algorithm, Greedy algorithm, Farthest Insertion algorithm, the Christofides algorithm and the Strip algorithm; and two Local Optimization Heuristics – 2-Opt algorithm and 3-Opt algorithms. This project enables the user to learn these algorithms easily and to get a deeper understanding of them. Learning these algorithms may help the user to improve basic problem-solving skills and to apply them to other similar problems. This project also takes advantage of the World Wide Web, so that it can be accessed conveniently through the Internet. The URL (Uniform and or Universal Resource Locator) of the home page that contains this project is http://a.cs.okstate.edu/~gyan/TSP.html.

To implement this project, knowledge of approximation algorithms for TSP, Java programming, HTML, as well as object-oriented design and programming was needed. Totally, there are 17 Java source code files and an HTML file in this project. Some statistical information of these files is shown in the following table.

File	Lines
Algorithm.java	191
Cf.java	572
Cfvertex.java	84
Display1.java	206
Display2.java	98
DrawCanvas.java	178
Edge.java	55
Fi.java	238
Greedy.java	366
IAlgorithm.java	21
Nn.java	81
Opt2.java	162
Opt3.java	119
Strip.java	358
TSP.html	114
TSP.java	470
TSPFrame.java	49
Vertex.java	57
Total	3419

Table 16. Statistics of the Program Files

5.2 Future Work

This project animated some traditional and popular approximation algorithms for Traveling Salesman problems. Some new algorithms, such as Genetic algorithms and Simulated Annealing have been developed in recent years; though they might not be standard ways to solve a TSP, they are quite valuable as new problem-solving techniques. In the future, these algorithms can be animated. Also, there are some very good exact algorithms to solve the Traveling Salesman Problem, and they can be animated in the future.

REFERENCES

[Bently and Saxe 1980] Bentley, J.L. and Saxe, J.B. (1980), An analysis of two heuristics for the euclidean traveling salesman problem, pp. 41-49 in *The Proceedings of the 18th Annual Allerton Conference on Communication, Control, and Computing* (October).

[Bland and Shallcross 1989] Bland, R.G. and Shallcross, D. F. (1989), Large traveling salesman problems arising in experiments in X-ray crystallography: A preliminary report on computation, *Operations Research Letters* 8, 125-128.

[Brady 1985] Brady, R.M. (1985), Optimization strategies gleaned from biological evolution, *Nature* **317**, 125-128.

[Chandler 1998] Chandler, J. P., Personal Communication.

[Christofides 1976] Christofides, N. (1976), Worst-case analysis of a new heuristic for the traveling salesman problem, *Report* 388, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, PA.

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

[Eiselt and Laporte1991] Eiselt, H.A. and Laporte, G. (1991), A combinatorial optimization problem arising in dartboard design, *Journal of the Operational Research Society* 42, 113-118.

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

[Garfinkel 1977] Garfinkel, R.S. (1977), Minimizing wallpaper waste, Part I: A class of traveling salesman problems, *Operational Research* 25, 741-751.

[Gosling, et al. 1996] Gosling, J., Joy, B. and Steele, G. (1996), *The Java Language Specification*, Addison Wesley Longman, Inc. Reading, Mass.

[Johnson and Papadimitriou 1985] Johnson, D.S. and Papadimitriou, C.H. (1985), Performance guarantees for heuristics, Chapter 5 of *The Traveling Salesman Problem*, John Wiley & Sons, New York, NY.

[Johnson 1990] Johnson, D.S. (1990), Local optimization and the traveling salesman problem, in *Automata, Languagees, and Programming, Lecture Notes in Computer Science 443*, Edited by Paterson M.S., Springer, Berlin.

[Johnson and McGeoch 1997] Johnson, D.S. and McGeoch, L.A. (1997), The traveling salesman problem: a case study, Chapter 8 of *Local Search in Combinatorial Optimization*, Edited by Aarts, E. and Lenstra, J.K., John Wiley & Sons Ltd.

[Kirkpatrick, S. et al 1983] Kirkpatrick, S. Gelatt, C.D. and Vecchi, M.P. (1983), Optimization by Simulated Annealing, *Science* **220**, 671-680.

[Lawler1976] Lawler, E.L. (1976), *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart, and Winston, New York.

[Lenstra and Rinnooy Kan, 1975] Lenstra, J.K. and Rinnooy Kan, A. H.G. (1975), Some simple applications of the traveling salesman problem, *Operational Research Quarterly* 26, 717-733.

[Lin 1956] Lin, S. (1965), Computer solutions of the traveling salesman problem. *Bell System Technical Journal* 44, 2245-2269.

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

[Reinelt 1989] Reinelt, G. (1989), Fast heuristics for large geometric traveling salesman problems, *Report No. 185*, Institut fur Mathematik, Universitat Augsburg.

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

APPENDIX A

GLOSSARY

Better	Tour A is better than tour B means tour a has a shorter tour length than that of tour B.
Euler tour	A simple cycle that passes through each edge exactly once
Feasible tour	A simple path in the graph, which goes through all vertices and each vertex in the path is distinct, except the first one and the last one are the same.
NP	All decision problems solvable by a non-deterministic algorithm in polynomial time
NP-hard	A problem is NP-hard if and only if the Satisfiability problem reduces to it.
Permutation	An ordered arrangement of the elements in a set.

2

VITA

YAN GU

Candidate for the Degree of

Master of Science

Thesis : ANIMATION OF APPROXIMATION ALGORITHMS FOR TRAVELING SALESMAN PROBLEM IN JAVA

Major Field : Computer Science

Biographical :

Personal Data : Born in Tianjin, People's Republic of China, March 12, 1969, the daughter of Weihuan Pan and Chuanqing Gu.

Education : Graduated from Tianjin No. 3 middle school, Tianjin, P. R. China, in July 1987; received Bachelor of Science Degree in Plant Physiology in July, 1991, and Master of Science Degree in Biochemistry in July,1994 from Nankai University, Tianjin, P. R. China; received Master of Science Degree in Biochemistry and Molecular Biology from Oklahoma State University, Stillwater, Oklahoma in May 1997; completed requirements for the Master of Science degree at Oklahoma State University in September, 1998.

Professional Experience: Teaching Assistant, Department of Computer Science, Oklahoma State University, January, 1998 to May, 1998; Research Assistant, Department of Biochemistry and Molecular Biology, Oklahoma State University, January, 1995 to May 1997; Instructor, Department of Biochemistry and Molecular Biology, Nankai University, July, 1994 to December, 1994.