Ant Colony Optimization and Rectilinear Steiner Minimal Trees

by

Anthony Curtis Blackman

B.Sc., University of the West Indies, 2003

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES
(Mathematics)

The University of British Columbia

April 2006

Abstract

This paper consists of two distinct parts. In the first part, we introduce the Rectilinear Steiner Minimal Tree ($RSMT$) problem and describe the Ant Colony algorithm which we use to construct the $RSMT$. The development of an Ant Colony algorithm called AntCol Steiner is described and we create a working program of the algorithm using the Sun Microsystems Java programming language (see appendix A to H for details). We investigate the effectiveness of our algorithm by using AntCol Steiner to construct and draw $RSMTs$ given a random set of terminals. In the second part, we look at an application of the Kirchoff's Matrix Tree theorem. We find the length of a Rectilinear Minimal Spanning Tree ($RMST$) spanning a set of random terminals. We verify the accuracy of our results from the Kirchoff's Matrix Tree algorithm using Prim's $RMST$ Algorithm. Our original plan was to develop an algorithm based on this theorem to determine the length of a $RSTM$. However, we were unable to modify the algorithm we used to find the length of the $RMST$, for the case of a $RSMT$.

Contents

Abstract ii

Contents iii

List of Tables vi

List of Figures vii

Acknowledgements ix

Dedication x

1 Introduction 1

1.1 Rectilinear Steiner Problem 1

1.2 Hanan's Grid 4

2 The Ant Colony Idea 6

2.1 Ant Colony Optimization (ACO) 8

2.2 Development of ACO Algorithms for the Steiner Problem 8

2.3 AntCol Steiner Algorithm Intuition 10

2.4 The Most Critical Methods of AntCol Steiner 11

2.4.1 The AntColSteiner(N, M) Method 11

2.4.2 The constructSteinerTreeByACO(terminals, grid, intensity, gridLimits) Method. 13

2.4.3 The AntMove(colony[m],info[1], info[2], id, grid, intensity, terminals, colony, antID, gridLimits) Method. 15
2.4.4 The pRune(sTree, grid, gridLimits) Method 16
2.5 Formulas used in the AntCol Steiner Algorithm. 18

3 Kirchoff's Matrix Steiner Tree Formula **21**
3.1 An Application of Kirchoff's Matrix-Tree Theorem 22
3.2 Kirchoff's Algorithm for Calculating the Minimal Length of a Tree . 25
3.2.1 Time Complexity of Calculating the Determinant 25
3.2.2 Algorithm Description 27
3.3 An Interesting Observation 29

4 Results **31**
4.1 Results from the AntCol Steiner Algorithm 31
4.2 Results from the Kirchoff's Matrix Tree Algorithm 32
4.2.1 Results from MST_via_MatrixTree_Nu(N) with N = 5 32

5 Conclusions **35**
5.1 AntCol Steiner Conclusion 35
5.2 Kirchoff's Matrix Tree Conclusion 36

Bibliography **38**

Appendix A Graphical Results Obtain from AntCol Steiner **40**

Appendix B Source Code for InputUI.java **45**

Appendix C Source Code for AntCol.java **62**

Appendix D Source Code for HananGrid.java **73**

Appendix E Source Code for TreeConstruct.java **83**

Appendix F Source Code for Ant.java 98

Appendix G Source Code for Utilities.java 127

Appendix H Source Code for Display.java 133

Appendix I Source Code for the MST_via_MatrixTree_Nu.m 136

Appendix J Prim's Algorithm using Matlab 142

List of Tables

3.1 The TREE_VERT data structure. 28

3.2 The VSET data structure corresponding to table 6. 28

4.1 AntCol Steiner results. 32

List of Figures

1.1 An example of a RSMT constucted using GeoSteiner 3.1 [14]. . . . 2

1.2 An example of a Hanan grid for a set T with $N = 20$ points. 5

2.1 Arrow thickness represents probability of ants choosing one path over another. (t=0) Foraging ant chooses the path based on a 50% probability; (t=1) Follower ants also choose their path on a 50% probability; (t=2) Ants that choose the shorter path get the food faster; (t=3) First forager is coming back to the nest, because more ants completed the shorter path, a greater pheromone concentration will attract the first ant to that path; (t=4) Following ants choose the shorter path with a higher probability as more and more ants choose the same path; (t=5) After some time of this autocatalytic process the probability of ants choosing the longer path is very small. 7

A.1 The Graphical User Interface of the AntCol Steiner Program. 41

A.2 This Screen show the Steiner Tree before and after Pruning. The pre-pruned tree is located in the bottom left hand corner of the figure. The pruned tree is on the right hand side of the figure. 42

A.3 Hanan Grid with M = 50 terminals which was generated using AntCol Steiner. On this grid the "x" represents the terminal vertices and the "o" represent the possible Steiner points. 43

A.4 The Steiner Tree with M = 50 terminals; Tree Length = 182 units and Total Time = 14016 milliseconds obtained after using the AntCol Steiner program on the Hanan Grid above A.3. On this tree the "x" represents the terminal vertices and the "o" represent the Steiner points. The parameter values are Alpha = 5, Beta = 1, Q = 1000, Rho = 0.1, Gamma = 1 and MAXLOOP = 30. 44

Acknowledgements

I would like to thank my supervisor, Prof. David Brydges for his guidance, his patience and for his many wonderful suggestions, corrections and remarks over the last few months. His advice and kind ways were always motivational and instrumental to my progress. Many thanks to Prof. Gordon Slade for proofreading this thesis and for his help and assistance reorganizing the thesis.

I am specially grateful to the University of British Columbia, Probability Group for their generous financial support over the years.

This paper is rightfully dedicated to my parents, Anthony, and Beverley Blackman and to my sister Kristle Blackman whom have all given me tremendous emotional support when I needed it most.

ANTHONY CURTIS BLACKMAN

The University of British Columbia
April 2006

To my family Anthony, Beverley and Kristle Blackman.

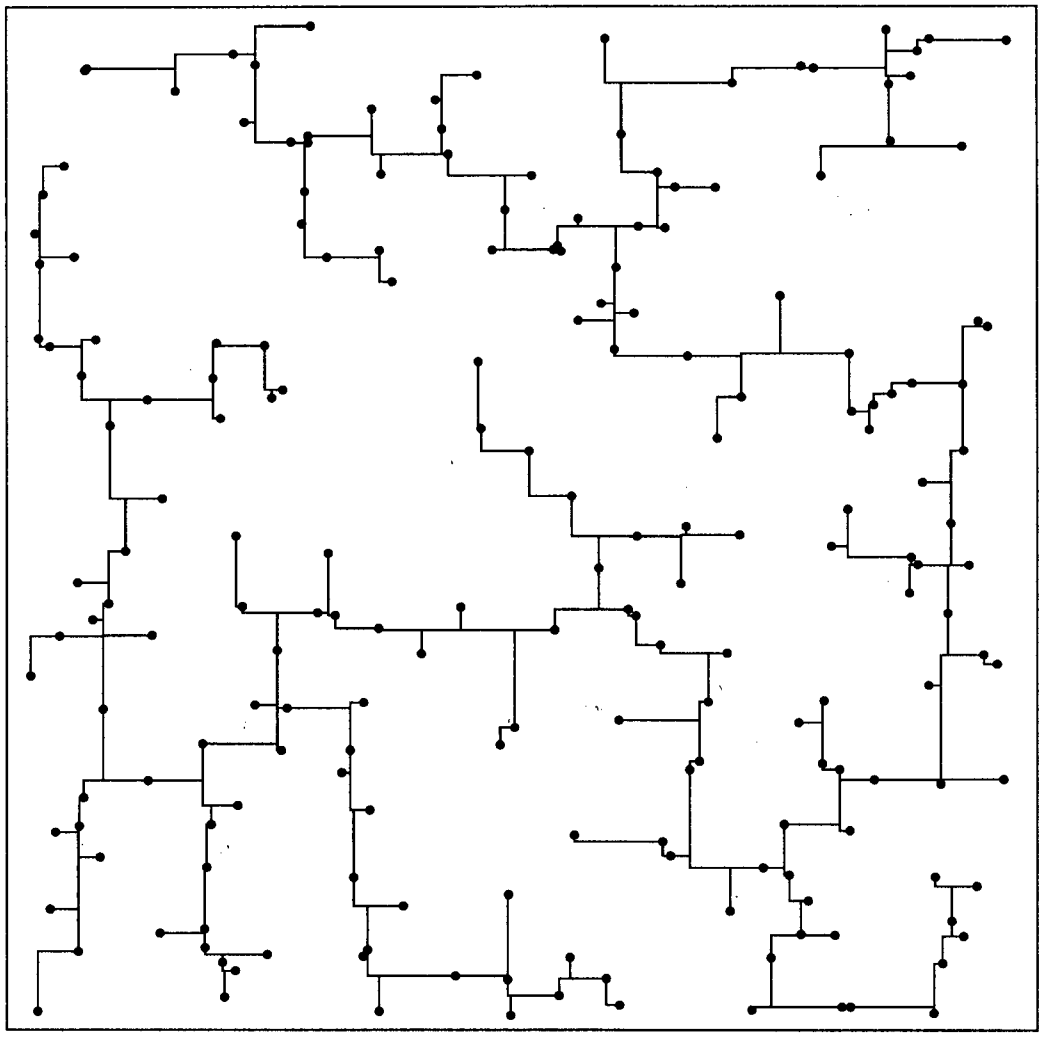
Chapter 1

Introduction

1.1 Rectilinear Steiner Problem

Connecting a set of points in the plane with a minimum length collection of vertical and horizontal wires is called the rectilinear Steiner problem. These spanning trees have been used often in the placement and global routing phases of circuit design since timing considerations require routing paths of minimal length. Optimum solutions to the problem have been investigated, but since the problem was proved to be NP-complete by Garey, M. R., and D. S. Johnson [7], only heuristic solutions are practical. The fastest exact rectilinear Steiner minimal tree (RSMT) algorithm is Geo-Steiner 3.1 created by David Warme, Pawel Winter, and Martin Zachariasen [14]. However, the main drawback to this algorithm is the length of time it takes to produce a solution. In figure 1.1 we illustrate a $RSMT$ which is a solution of GeoSteiner 3.1 [14].

An algorithm based on Ant Colony Optimization (ACO) was developed by Gurdip Singh, Sanjoy Das, Shekhar V. Gosavi, and Sandeep Pujar in [4]. Then in an attempt to find an algorithm which has faster convergence times than Geo-Steiner 3.1, an ACO algorithm based on the ideas of [4] called ACO-Steiner [9] was created. The algorithm shows significant improvements in convergence times for large values of M. However, these improvements were achieved by sacrificing the accuracy in



Steiner Minimal Tree: 200 points, length = 104178, 3.47 seconds

Figure 1.1: An example of a RSMT constucted using GeoSteiner 3.1 [14].

determining the length of the $RSMT$. These conclusions can be seen from Table 1 of [9].

Ants live in colonies and have evolved to exhibit very complex patterns of social interaction. Such interactions are clearly seen in the foraging strategy of ants. Despite the extremely simplistic behavior of individual ants they can communicate with one another through secretions called pheromone. This cooperative activity of the ants in a nest gives rise to an emergent phenomenon known as swarm intelligence [2]. Ant Colony Optimization (ACO) algorithms are a class of algorithms that mimic the cooperative behavior of real ants to achieve complex computations. Ant colony optimization was originally introduced as a metaheuristic for the well-known traveling salesman problem (TSP), which is a path based optimization problem. This problem is proven to be NP-complete, which is a subset of a class of difficult optimization problems that are not solvable in polynomial time (unless P=NP). Since an exponential time algorithm is infeasible for larger scale problems in class NP, much research has focused on applying stochastic optimization algorithms such as genetic algorithms and simulated annealing to obtain good (but not necessarily globally optimal) solutions. The ant colony approach is a stochastic algorithm with some ability to learn from previous attempts to solve the problem.

Definition 1.1.1 *A tree is said to be rectilinear if all of its edges are horizontal or vertical.*

Definition 1.1.2 *The RSMT problem is described as follows [9]. Given a set T of M points called terminals in the plane, find a set S of additional points called Steiner points such that the length of a rectilinear minimal spanning tree with vertex set $T \cup S$ is minimized.*

Definition 1.1.3 *The cost between vertex i and vertex j is $c(i,j)$, which is the Manhattan distance between vertex i and j.*

Definition 1.1.4 *The Manhattan distance function computes the distance that would be traveled to get from one data point to the other if a grid-like path is followed. The Manhattan distance between a point $i = (i_1, i_2)$ and a point $j = (j_1, j_2)$ is:*

$$c(i,j) = \mid i_1 - j_1 \mid + \mid i_2 - j_2 \mid \tag{1.1}$$

1.2 Hanan's Grid

Definition 1.2.1 *Given a set T of M points in the plane, the Hanan grid [8] is obtained by constructing a horizontal and a vertical line through each point in T.*

This grid is very helpful since the following theorem by Hanan [8] states that all of the possible Steiner points S can be found at intersections of this grid. This can be stated more precisely as follows:

Theorem 1.2.1 *A shortest rectilinear Steiner spanning tree over a set of points exists on the Hanan grid induced by the points.*

From looking at the figure 1.2 this theorem is intuitive since the edges of any rectilinear spanning tree can only coincide with the edges of the grid. That is every possible rectilinear spanning tree is a subset of the Hanan grid.

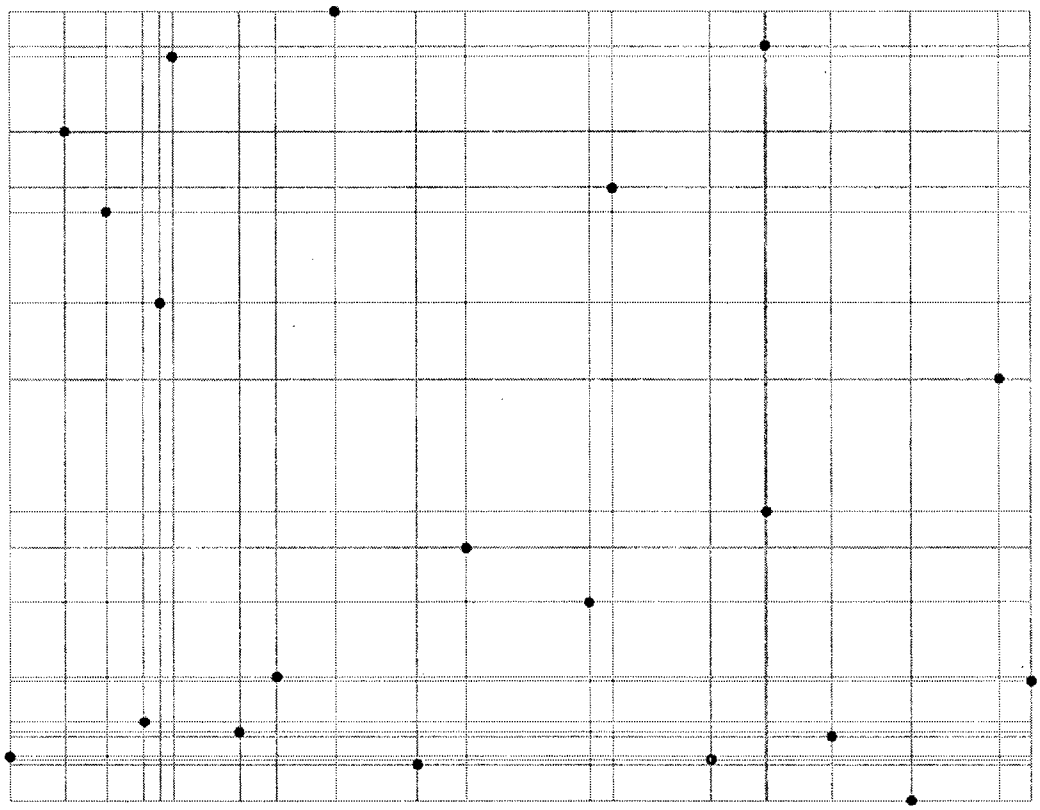


Figure 1.2: An example of a Hanan grid for a set T with $N = 20$ points.

Chapter 2

The Ant Colony Idea

The Ant System algorithm is biologically inspired by the foraging pattern of ants. Natural ants when following a pheromone trail decide where to go next based on the concentration of pheromone in a given direction. Deneubourg [5] performed an experiment with real world ants that consists of a trail leading from the ant's nest to a food source. Along the way, at certain points the trail branched into two paths of equal length, then unified into one trail again shortly after. In a controlled environment, ants in search of food would follow that trail and then would have to decide which branch to take. The first ant has no additional information on which to base its decision, so it just chooses any one of the two at random. Other ants following will also choose with roughly equal probability. When the first ant is coming back from the food location towards the nest, it again will randomly choose one of the trails because the amount of pheromone in the two trails will be very similar. The next experiment, known as double bridge made things very clear. Again, the ants are in an artificial controlled environment where there is a path from their nest to a food source. This time, there are two bifurcations in sequence as in figure 2.1, but unlike the first experiment, one branching path is shorter than the others. Initially the situation will be the same as the first experiment, but when the first ant is returning from food to nest, the concentration of pheromone

in the shorter path will be stronger, simply because the flow of ants using that path was greater due to the length difference. Ants who chose the shorter path will get to the unification point sooner that the ones who chose the longer paths. This autocatalytic process starts to repeat over and over, and soon almost all ants will be choosing the shorter path over the longer ones.

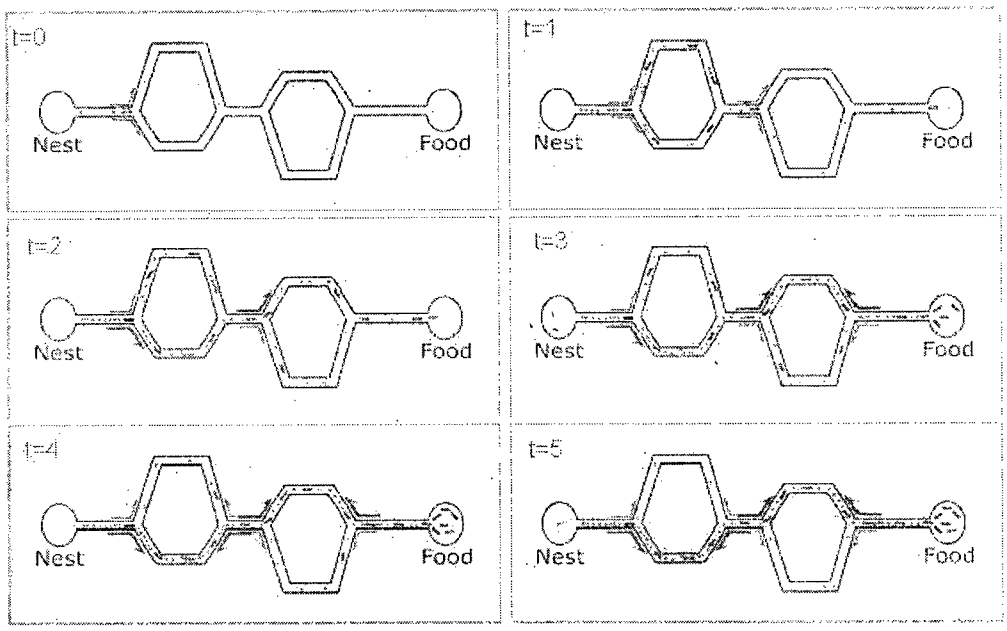


Figure 2.1: Arrow thickness represents probability of ants choosing one path over another. (t=0) Foraging ant chooses the path based on a 50% probability; (t=1) Follower ants also choose their path on a 50% probability; (t=2) Ants that choose the shorter path get the food faster; (t=3) First forager is coming back to the nest, because more ants completed the shorter path, a greater pheromone concentration will attract the first ant to that path; (t=4) Following ants choose the shorter path with a higher probability as more and more ants choose the same path; (t=5) After some time of this autocatalytic process the probability of ants choosing the longer path is very small.

2.1 Ant Colony Optimization (ACO)

From the conclusion of the experiments of Deneubourg an algorithm based on these observation seems to be a natural approach to obtaining an approximate time efficient solution to the Steiner problem. Algorithms which mimic this natural behavior of ants are called Ant Colony Optimization meta-heuristic algorithms.

2.2 Development of ACO Algorithms for the Steiner Problem

The Ant Colony Optimization algorithms employs a cooperating colony of ants in an attempt to find good solutions to difficult discrete optimization problems. Cooperation is a key design component of ACO algorithms. In these algorithms the artificial ants share the computational resources and communicate among each other indirectly. Good solutions are an emergent property of the artificial ants cooperative interaction.

Artificial ants have a double nature. First, they are an abstraction of real ants found in nature and of their behavior. On the other hand, artificial ants have also been enriched with some capabilities which are not found among their natural counterparts. Since the artificial ants are employed in software to solve difficult discrete optimization problems, it is reasonable to give the ants some capabilities, which while not found in real ant behavior, make them more efficient and effective. We will comment on this more later.

First, in a paper called "Ant Colony Algorithms for Steiner Trees: An Application to Routing in Sensor Networks" an algorithm based on Ant Colony Optimization was developed by Gurdip Singh, Sanjoy Das, Shekhar V. Gosavi, and Sandeep Pujar. This algorithm can be found in [4].

In this paper, the authors verified their results using various problem instances taken from Beasley [1]. The test cases described in problem set B, Beasley

[1], which consisted of randomly generated graphs whose edges were assigned weights between 1 and 10, were used. The test cases chosen consisted of instances with 50 to 100 nodes, and up to 200 edges. These problem instances all had well known optimal solutions and the authors used Beasley's results to determine the accuracy of their algorithm.

In [4] the authors used Matlab to implement three strategies for selecting the ants to be moved between vertices. These strategies were:

1. Fixed sequence strategy: Here the ants are picked in a fixed sequence, for reassignment into a different vertex. After the last ant is moved, the algorithm is reset to pick the first ant in the fixed sequence.

2. Shuffled sequence strategy: Here the algorithm also iterates through the ant set. However, after each ant was moved to a different location exactly once, the algorithm shuffles the order of the ants randomly before beginning all over again.

3. Random strategy: In comparison to the other methods, this approach is the most random way to move ants. No ordering is maintained, and an ant is picked up for movement from the set of all existing ants in a random manner with uniform probability.

From among these strategies the Random approach gave the best results when compared to [1] problem set B.

Secondly, an Ant Colony Optimization Steiner Algorithm called ACO-Steiner was developed by Hu, Jing, Hong, Feng, Hu and Yan [9]. In this case they looked at the RSMT problem on the Hanan's Grid. They used C++ to implement their algorithm and used the Random strategy described in [4] to select the ants to be moved between vertices. The experimental results of ACO-Steiner were compared with Geo-Steiner 3.1 and showed that ACO-Steiner could produce shorter running times while keeping comparable high performance.

2.3 AntCol Steiner Algorithm Intuition

After seeing the results of applying the ACO to the Steiner problem in [4] we decided to develop an ACO algorithm called AntCol Steiner using Java as the language of implementation. AntCol Steiner is a hybrid between the ideas of ACO-Steiner and the algorithm found in [4]. Before we get into the details of the AntCol Steiner algorithm. I will present an intuitive overview of how the algorithm really works. Note that the ants in our program use information that would not be available to more realistic ants such as the distance to the closest trail of another ant.

The following steps, which build a tree, are repeated MAXLOOP times:

Every ant is initially at a terminal. An ant is selected at random and moved until it dies as described below. Then a remaining ant is selected at random and so on until only one ant is left.

Each ant has a "tabu list", called **"tour"**, of points in the Hanan grid that it is forbidden to move to. Initially this list contains only the terminal where the ant starts. Each Hanan grid point visited by the ant is copied into the ant's tabu list. When an ant first encounters a point in the tabu list of another ant, it appends its tabu list into the other ant's tabu list and it dies.

The moving ant deterministically moves to a neighboring grid point according to the formula (2.4) unless it is trapped. Formula (2.4) requires extensive explanation which will be given later, but the idea is that one selects the neighbor in the Hanan grid move according pheromone level and according to how close the neighbor is to the tabu list of some other ant, because the tree is short if the ants die quickly.

If the moving ant is trapped then it is repeatedly relocated to a randomly chosen point in its tabu list until it is no longer trapped. In the program this procedure is called "de-confused".

After all but one ant has died the tree is pruned in order to remove dead-ends created when ants get trapped. Next, the pheromone on the edges of the Hanan

grid is updated.

If the length of the pruned tree is smaller that that of the best previously found tree then the previously found tree is replaced by the new one.

2.4 The Most Critical Methods of AntCol Steiner

In this section we describe the core methods of the AntCol Steiner program. These methods are AntColSteiner, constructSteinerTreeByACO, AntMove and pRune. AntColSteiner is used to select the best of all possible trees. constructSteinerTree-ByACO implements the ant selection strategy and constructs the tree by merging the trails of each of the selected ants when they meet. AntMove is used to determine which direction the ant should move via the Hanan grid. Finally, the pRune method removes all non terminal leaves from the tree obtained from constructSteinerTree-ByACO.

2.4.1 The AntColSteiner(N, M) Method

INPUT ARGUMENTS:

N : This is the dimensions of a square grid. M : This is the number of terminal points that are randomly generated and placed inside the square array.

OUTPUT A Rectilinear Steiner Tree

1. BEGIN.
2. An N by N array of string type is initialized with blank spaces.
3. M random vertices are generated. They have the form (r, c) where r represents row r and c represents column c of the grid. r and c were chosen to be even numbers so that each point in the grid was at least one space away from the other points. The only reason vertices were placed in rows r and columns c with even indexes was to make our output easier to understand.

4. At each vertex (r, c) the letter x is placed on the grid to indicate that a terminal vertex is created.

5. Using the terminal vertices a Hanan Grid is generated. During the construction of the grid each time a vertical edge and a horizontal edge crossed, the letter o is placed on the grid to represent a possible Steiner point.

6. An equal amount of pheromone is placed on each edge of the Hanan Grid. This represents the initial trail intensity.

7. An ant data structure is created and M instances of the data structure are declared and initialized. These are our artificial ants.

8. Initialize the **bestSteinerTree = constructSteinerTreeByACO(terminals, grid, intensity, gridLimits)** see the description of this method below.

9. **bestSlength = getTLength()** This method gets the length of the Steiner tree just generated.

10. BEGIN WHILE

11. while(count < MAXLOOP)

    - FOR each edge in the Hanan grid
      - IF
        * The ant walked on edge of the grid during the previous iterations, update the pheromone intensity along that edge using **updateIntensity(sLength, row, col, intensity)**
      - ELSE reduce the level of pheromone intensity.
      - ENDIF
    - ENDFOR

12. **steinerTree = constructSteinerTreeByACO(terminals, grid, intensity, gridLimits)**

13. **sLength = getTLength()** This method returns the length of the Steiner tree just generated.

14. **indicate = chooseBestSol(bestSlength, sLength)** returns true if old tree is still the best.

    - IF **(indicate)**
      - keep the bestSteinerTree.
    - ELSE
      - copy the content of steinerTree into bestSteinerTree.
    - ENDIF

15. count = count + 1.

16. END WHILE.

17. After MAXLOOP iterations bestSteinerTree was displayed.

18. END.

2.4.2 The constructSteinerTreeByACO(terminals, grid, intensity, gridLimits) Method.

INPUT ARGUMENTS:

terminals: This is a two dimensional integer array used to store all of the vertices in such a way that $(terminals[i][0], terminals[i][1]) = (r, c)_i$ where i represents the ith vertex that was randomly generated. **gridLimits:** This is an integer array used to store the border values of the grid in the following manner: **gridLimits[0] = xmax** This is the value of the last row of the Hanan's grid. **gridLimits[1] = ymax** This is the value of the last column of the Hanan's grid. **gridLimits[2] = xmin** This is the value of the first row of the Hanan's grid. **gridLimits[3] = ymin** This is the

value of the first column of the Hanan's grid. **intensity:** This is the an array that stores the intensity values for all of edges in the Hanan's grid.

OUTPUT A Rectilinear Steiner Tree

1. BEGIN
2. Place an ant on each of the M terminal vertices (r, c) of the Hanan's grid using the **placeAnt(tool, colony[i],terminals[i][0], terminals[i][1], i)** method for $i = 1$ to M.
3. WHILE
   - while(ant number greater than one)
   - choose one of the ants sitting on the vertices of the grid randomly.
   - Move the chosen ant to one of its neighboring vertices using the **info = AntMove(tool, colony[m],info[1], info[2], id, grid, intensity, terminals, colony, antID, gridLimits)** method.
   - WHILE The ant k is not at a vertex in the tabu list of another ant
     - Continue moving him using the **info = AntMove(tool, colony[m],info[1], info[2], id, grid, intensity, terminals, colony, antID, gridLimits)** method.
   - END WHILE
   - IF The ant k met a vertex in the tabu list of ant r.
     - The tabu list of ant r is expanded so as to include the tabu list of ant k.
     - Ant k will be killed.
     - ant number = ant number -1;
     - Ant r will be relocated to another vertex in its own tabu-list.
   - END IF

4. END WHILE

5. **sTree = pRune(sTree, grid, gridLimits)** This method was used to remove all leaves that were not terminals.

6. **Return sTree**

7. END

2.4.3 The AntMove(colony[m],info[1], info[2], id, grid, intensity, terminals, colony, antID, gridLimits) Method.

INPUT ARGUMENTS:

info[1] stores the current row the ant occupies. **info[2]** stores the current column the ant occupies. **colony** This stores the entire nest of ants and their personal information such as nodes visited. **colony[m]** This is the current ant that has been selected to move. **id** This is the indicator that was used to determine if an ant met the tabu list of another ant or not. If the ant met another ant **id** would store the identification number of the ant it met. Else, **id** would have the value of -1. **antID** This is an array which indicates the mortal status of the ants in the colony. If **antID[i]** $= -1$ this means ant **i** is dead. If **antID[i]** $= 1$ this means ant **i** is alive.

1. BEGIN

2. The ant who was chosen, will "stand up" and look around at all of its immediate nearest neighbors vertices (possible Steiner points as well as terminal vertices). An ant can have at most four such neighbors on the Hanan Grid.

- IF any of its neighbors are in the tabu list (**tour**) then it is not allowed to visit that neighbor again. That means that the weight of visiting that vertex is 0.
- ELSE The ant will calculate the weight to move to that vertex via the edge on the Hanan Grid by using a formula (2.4) which takes into consideration the pheromone intensity (2.2) levels and the ant's desire (2.1) to go to its neighbor.
- ENDIF

3. IF

   - all the weight to visit its neighboring vertices is zero then the ant is trapped. In this case we have to "de-confuse" the ant. De-confuse means to repeatedly place it at a random location in its tabu list until it is not trapped anymore. This de-confusion was executed using the **deconfuse(m,terminals, grid, antIndex)** method.

4. ELSE

   - There will be positive weight to visit one of its neighbors. So after all the weights of visiting its possible neighbors have been calculated the ant moves to the neighbor with the highest weight.

5. ENDIF

6. **Return info**

7. END

2.4.4 The pRune(sTree, grid, gridLimits) Method

INPUT ARGUMENTS:

A unpruned Rectilinear Steiner Tree called sTree.

During the execution of the program there are occasions when the ant has nowhere to go because the weight to visit all of its neighbors is zero. In this case the ant is said to be confused and has to be de-confused. The de-confuse method repeatedly places it on a random vertex in its tabu list until the ant is no longer confused. What happens during this procedure is that the trail which caused the ant to be trapped is left on the grid and is a dead end trail. The end of the trail is a vertex which is not in the tabu list of another ant. This is because if it was then the ant would have met another ant, died and would not have been confused. Therefore the only other possibility is that the vertex has to be a possible Steiner point. This fact plus the fact that the vertex is at the end of a dead end trail means that it is a non terminal leaf in the tree. These non terminal leaves have to be trimmed off the tree. This process of trimming off all of the non terminal leaves and their trails is called pruning. The pRune method is used to perform the pruning in the AntCol Steiner program.

OUTPUT A Pruned Rectilinear Steiner Tree called sTree.

1. BEGIN
2. FOR loops = 0 to loops = NSquared
   - FOR row = gridLimits[2] to gridLimits[0]
     - FOR col = gridLimits[3] to gridLimits[1]
       - IF grid[row][col] is NOT "x" AND sTree[row][col] is NOT " "
         - calculate the degree of the sTree element at position (row, col)
         - IF the degree of the element at position (row, col) of sTree is equal to one then delete sTree[row][col]
         - ENDIF
       - ENDIF

       col = col + 1

- ENDFOR

row = row + 1

- ENDFOR

loops = loops + 1

3. ENDFOR

4. **Return sTree**

5. END

Remark 2.4.1 *Since all of the ants were leaving pheromone on the edges of the Hanan grid they crossed, there will be edges which have strong pheromone levels and edges which do not have a high concentration of pheromone. Therefore the edges with the highest pheromone levels will have a higher probability of being crossed again thus further increasing the probability of being crossed in the future iteration of the algorithm. This means that the more often a path is crossed the more likely it will be to be crossed in the future. The convergence theory of Ant Colony algorithms can be found in section 4.3 of [6], but the results there do not claim that the algorithm converges to a minimal tree.*

2.5 Formulas used in the AntCol Steiner Algorithm.

In AntCol Steiner, we generate the Hanan Grid of the terminal set T. Then we place ants on each element of T. An ant i then determines a new vertex j to visit via an edge of the Hanan grid. The choice is based on a higher value $p_{i,j}$, which is a trade-off between the desirability and the trail intensity. Each ant maintains its own tabu-list, which records the vertices it visits to avoid revisiting them again. Then in the case where ant Γ meets ant Ω, ant Γ dies, and adds the vertices in its tabu-list into the tabu-list of ant Ω. After every movement of an ant the trail of the ant is

also recorded in a data structure called globalView. This data structure allow us to see the interaction of the ants while they are building the RSMT. Each ant in the globalView data structure is identified via its identification number. Therefore, the trail of ant 2 would be displayed as sequence of 2's on the path of the trail in the globalView data structure, while the trail of ant 3 would be displayed as a sequence of 3's on the path of the trail in the globalView data structure and so on. At the point the ant Γ meets ant Ω, ant Γ dies and the trail labeled Γ in the globalView data structure would be merged with Ω's trail in the globalView data structure. Thus, all Γs will be changed to Ωs in the globalView data structure to reflect Ω's new trail. The trail intensity is updated at the end of each iteration of MAXLOOP. The globalView data structure eventually contains the ant identification number on the trail of the only surviving ant. This identification trail is used to display the RMST before pruning.

Let j be a neighbor of i in the Hanan grid. Given an ant m on vertex i, the desirability of vertex j is:

$$\eta_{i,j}^{m} = \frac{1}{[c(i,j) + \gamma \cdot \psi_{i,j}^{m}]} \tag{2.1}$$

where γ is a constant, $\psi_{i,j}^{m}$ is the distance from vertex i to the nearest vertex in the tabu-list of another ant. This makes the current ant join into others as quickly as possible. The superscript m is there because $\psi_{i,j}^{m}$ depends on the previous ant trails.

The updating of the trail intensity in the Hanan edge (i,j) is defined as follows:

$$\tau_{i,j}(t+1) = (1-\rho) \cdot \tau_{i,j}(t) + \rho \cdot \Delta\tau_{i,j}(t) \tag{2.2}$$

where ρ is a constant, representing the pheromone evaporation parameter. t, which is called **count**, is the variable that ranges between 1 and MAXLOOP in our earlier summary on page 10. The increments of updating is given by the following formula:

$$\Delta\tau_{i,j}(t) = \begin{cases} \frac{Q}{c(S(t))} & \text{if } (i,j) \in E(t) \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

where $c(S(t))$ is the total cost of the current Steiner tree $S(t)$. $E(t)$ is the edge set of $S(t)$ and Q is a parameter constant that specifies the amount of pheromone the surviving ant has to distribute through its trail.

The weight of an ant at i to move on edge (i,j) is defined as follows:

$$p_{i,j}(t) = \begin{cases} \frac{[\tau_{i,j}(t)]^{\alpha}[\eta_{i,j}^{m}]^{\beta}}{\sum_k [\tau_{i,k}(t)]^{\alpha}[\eta_{i,j}^{m}]^{\beta}} & \text{if } j \in A \text{ and } k \notin \text{tabu-list}(m) \\ 0 & \text{otherwise} \end{cases} \tag{2.4}$$

where A is the set making up of all vertices which are connected with i and is not in the tabu-list of ant m. The parameters α and β control the relative importance of each variable $\tau_{i,j}$ and $\eta_{i,j}^{m}$ in (2.4). The results of this algorithm can be seen in section 4.1.

Chapter 3

Kirchoff's Matrix Steiner Tree Formula

This section describes an algorithm to calculate the length of a minimal spanning tree on N vertices using the matrix tree theorem. The algorithm is $O(N^3)$ but not as fast as the standard Prim's algorithm [13] which can be shown to run in time which is $O(E \ln N)$ where E is the number of edges.

The reason we looked at this method was that we hoped that we could adapt it to find the length of the minimal Steiner tree without calculating the Steiner tree or the Steiner points and perhaps thereby avoid the NP completeness problem, but we have not so far found a formula in terms determinants that properly accounts for Steiner points. For this reason the content in this chapter is not closely related to the previous chapters of this thesis and only represents a possible future direction.

The matrix tree theorem is as follows. Let $A_{\alpha\beta}$ be a symmetric $N \times N$ matrix whose row sums are zero except for the first row which sums to one.

Theorem 3.0.1 *Kirchoff's matrix-tree theorem [12].*

$$\det A = \sum_{T} \prod_{\alpha,\beta \in T} A_{\alpha\beta}$$

where T is summed over all tree graphs with vertex set $\{1, 2, \cdots, N\}$ and $\alpha, \beta \in T$ means $\{\alpha, \beta\}$ is in the edge set of T.

3.1 An Application of Kirchoff's Matrix-Tree Theorem

Let $Vertex\_set = \{K_\alpha\}$ be a finite set of points in the plane. These are the terminals. We are about to describe a way to calculate the length of a minimal spanning tree. Note there are no Steiner points.

Let K_α and $K_\beta \in Vertex\_set$. If $K_\alpha = (i_\alpha, j_\alpha)$ and $K_\beta = (i_\beta, j_\beta)$ for any $\alpha, \beta = 1, 2, \cdots, N$, the Manhattan distance $c(K_\alpha, K_\beta)$ between K_α and K_β is given by definition 1.1.4.

Define the matrix $\mathbf{A(m)}$ by

$$A_{\alpha\beta}(m) = \begin{cases} -e^{-m \cdot c(K_\alpha, K_\beta)} & \text{if } \alpha \neq \beta \\ \sum_{K_\gamma \neq K_\alpha} e^{-m \cdot c(K_\alpha, K_\gamma)} + \delta_1(\alpha)\delta_1(\beta) & \text{if } \alpha = \beta. \end{cases} \tag{3.1}$$

From this definition $\mathbf{A(m)}$ is a symmetric matrix of the form

$$\mathbf{A(m)} = \begin{pmatrix} A_{11}(m) & A_{12}(m) & \dots & A_{1N}(m) \\ A_{21}(m) & A_{22}(m) & \dots & A_{2N}(m) \\ \vdots & \vdots & \ddots & \vdots \\ A_{N1}(m) & A_{N2}(m) & \dots & A_{NN}(m) \end{pmatrix}$$

with $A_{ij}(m) = A_{ji}(m)$, that is $(\mathbf{A(m)})^T = \mathbf{A(m)}$. Further all of the row sums of $A(m)$ are zero except for the first row which sums to one. That is $\sum_{k=1}^{N} A_{1k}(m) = 1$ and $\sum_{k=1}^{N} A_{jk}(m) = 0$ for each row $j = 2, \cdots, N$.

Using this definition and the Matrix-Tree Theorem 3.0.1 we obtain.

$$\begin{aligned}\det A(m) &= \sum_T \prod_{\alpha,\beta\in T} e^{-m\cdot c(K_\alpha,K_\beta)} \\ &= \sum_T e^{-m\cdot\left[\sum_{\alpha,\beta\in T} c(K_\alpha,K_\beta)\right]} \\ &= \sum_T e^{-m\cdot(Total\_length(T))}\end{aligned}$$

Further, we can fix N and take $m \to \infty$ and consider $\frac{1}{m}\left[\ln\det \mathbf{A(m)}\right]$. We obtain

$$\begin{aligned}\frac{1}{m}\left[\ln\det \mathbf{A(m)}\right] &= \frac{1}{m}\left[\ln\sum_T e^{-m\cdot(Total\_length(T))}\right] \\ &\sim \frac{1}{m}\left[\ln e^{-m\cdot(Minimum\_length(T))}\right] \\ &= -Minimum\_length(T)\end{aligned}$$

From this we have a formula for calculating the length of the minimum spanning tree T via $\det \mathbf{A(m)}$.

Example 3.1.1 *Consider the graph G which has three vertices labelled V_1, V_2 and V_3. Let the rectilinear distance between each vertex be defined as follows. $c(V_1, V_2) = 2$, $c(V_1, V_3) = 3$ and $c(V_2, V_3) = 6$ then the length of the minimum spanning tree of G is clearly $c(V_1, V_2) + c(V_1, V_3) = 2 + 3 = 5$. I will demonstrate the Matrix-Tree theorem by calculating the length of the minimal spanning tree using (3.1).*

The distance matrix $\mathbf{D}$ for the example above is given below. In this matrix $\mathbf{D}$ an element in row i and column j represents the rectilinear distance between vertex V_i and vertex V_j.

$$\mathbf{D} = \begin{pmatrix} 0 & 2 & 3 \\ 2 & 0 & 6 \\ 3 & 6 & 0 \end{pmatrix}$$

From this we obtain:

$$\mathbf{A(m)} = \begin{pmatrix} (1+e^{-2m}+e^{-3m}) & -e^{-2m} & -e^{-3m} \\ -e^{-2m} & (e^{-2m}+e^{-6m}) & -e^{-6m} \\ -e^{-3m} & -e^{-6m} & (e^{-3m}+e^{-6m}) \end{pmatrix}.$$

Therefore,

$$\begin{aligned} \det \mathbf{A(m)} &= (1+e^{-2m}+e^{-3m})\Big[(e^{-2m}+e^{-6m})(e^{-3m}+e^{-6m}) - e^{-12m}\Big] \\ &\quad -e^{-2m}\Big[-e^{-9m} - e^{-2m}(e^{-3m}+e^{-6m})\Big] \\ &\quad -e^{-3m}\Big[-e^{-8m} - e^{-3m}(e^{-2m}+e^{-6m})\Big] \\ &= e^{-5m} + 2e^{-7m} + e^{-8m} + e^{-9m} + 2e^{-10m} + 4e^{-11m} \\ &= e^{-5m}\Big[1 + 2e^{-2m} + e^{-3m} + e^{-4m} + 2e^{-5m} + 4e^{-6m}\Big]. \end{aligned}$$

Now if we take natural logs on both sides we obtain,

$$\ln\Big[\det \mathbf{A(m)}\Big] = -5m + \ln\Big[1 + 2e^{-2m} + e^{-3m} + e^{-4m} + 2e^{-5m} + 4e^{-6m}\Big]$$

which implies that,

$$\frac{1}{m}\Big[\ln\det \mathbf{A(m)}\Big] = -5 + \frac{\ln\Big[1 + 2e^{-2m} + e^{-3m} + e^{-4m} + 2e^{-5m} + 4e^{-6m}\Big]}{m}.$$

Therefore we conclude that,

$$\frac{1}{m}\Big[\ln\det \mathbf{A(m)}\Big] \to -5$$

as $m \to \infty$. So the absolute value of our limit agrees with the length of the minimal spanning tree as expected.

Now that we have an idea that this technique actually work we will now develop an algorithm to compute the length of the minimal spanning tree of a graph G which has n randomly generated vertices.

3.2 Kirchoff's Algorithm for Calculating the Minimal Length of a Tree

In the previous section we saw a formula that related the minimal length of T to $\det \mathbf{A}(\mathbf{m})$. Now we verify that $\det \mathbf{A}(\mathbf{m})$ can be computed in polynomial time. In fact it can be shown that there exists an algorithm that will calculate the $\det \mathbf{A}(\mathbf{m})$ with time complexity $O(n^3)$. The main idea behind this algorithm is Gaussian Elimination.

3.2.1 Time Complexity of Calculating the Determinant

For simplicity in notation of the proof we will write $\det(A)$ instead of $\det \mathbf{A}(\mathbf{m})$. The idea is to evaluate $\det(A)$ by reducing the matrix A to an upper-triangular form, in which case the determinant is just the product of elements on the diagonal. The general algorithm is, for an $n \times n$ matrix:

FOR $i = 2 \cdots n$ row

FOR $j = 1 \cdots i-1$ row above

FOR $k = j \cdots n$ column to be reduced

$A_{jk} := A_{ik} - \frac{A_{ij}}{A_{jj}} \cdot A_{jk}$ (consider this to take constant time K).

In practice, we need to take precautions against the situation in which $A_{jj} = 0$, but this does not change the order of the algorithm, so will not be considered here. Once a matrix is in upper-triangular form, its determinant is just the product of elements on the diagonal $\det(A) = \prod_i A_{ii}$, so it can be evaluated with $(n-1)$ multiplications which takes order $O(n)$ time.

Theorem 3.2.1 *The cost of calculating det(A) for a general $n \times n$ matrix using Gaussian Elimination is $O(n^3)$.*

Proof. Let $C(n)$ be the cost to calculate the *det*(A) for a general $n \times n$ matrix A then,

$$
\begin{aligned}
C(n) &= \sum_{i=2}^{n}\sum_{j=1}^{i-1}\sum_{k=j}^{n} K \\
&= K\sum_{i=2}^{n}\sum_{j=1}^{i-1}(n+1-j) \\
&= K\sum_{i=2}^{n}\left[(n+1)(i-1) - \sum_{j=1}^{i-1} j\right] \\
&= K\sum_{i=2}^{n}\left[(n+1)(i-1) - \frac{i(i-1)}{2}\right] \\
&= K\sum_{i=1}^{n}\left[(n+1)(i-1) - \frac{i(i-1)}{2}\right] \\
&= \frac{K}{2}\sum_{i=1}^{n}\left[-i^2 - (2n+3)i - 2(n+1)\right] \\
&= \frac{K}{2}\left[-\frac{n(n+1)(2n+1)}{6} + \frac{n(n+1)(2n+3)}{2} - 2n(n+1)\right] \\
&= K\left[\frac{n(n+1)}{2}\right]\left[-\frac{1+2n}{6} + \frac{2n+3}{2} - 2\right] \\
&= K\left[\frac{n(n+1)}{2}\right]\left[\frac{4(n-1)}{6}\right] \\
&= K\left[\frac{2n(n+1)(n-1)}{6}\right] \\
&= K\left[\frac{n(n^2-1)}{3}\right] \in O(n^3)
\end{aligned}
$$

Since the cost of now evaluating the determinant of the upper-triangular matrix is just $O(n)$, it is correct to say that the cost $C(n)$ of the whole procedure is $O(n^3)$.

Remark 3.2.1 *In [11] there are algorithms which can calculate the determinant of a matrix A faster than $O(n^3)$.*

3.2.2 Algorithm Description

In this section, a detailed algorithm for determining the length of a minimum spanning tree T with N vertices is outlined. The algorithm is based on Kirchoff's Matrix-Tree theory and has a time complexity of $O(N^3)$.

1. BEGIN
2. Create a $N \times 3$ array and call it $VSET$ ($Vertex\_Set$).
3. Set all of the elements of array $VSET$ to zero.
4. Create two arrays of size $N \times N$. Call the first one $MTREE$ ($Matrix\_Tree$) and the other one $TREE\_VERT$ ($Tree\_Vertex$).
5. Set all of the elements of $MTREE$ and $TREE\_VERT$ to zero.
6. FOR $k = 1$ to N
   - Generate a random number between 1 and N.
   - Store the number in i.
   - Generate a random number between 1 and N.
   - Store the number in j.
   - At position k insert, value of k in column 1; value of i in column 2 and the value of j in column 3 of $VSET$.
   - Put a 1 at position (i, j) in $TREE\_VERT$

   ENDFOR

Example 3.2.1 *Consider the arrays below which illustrates the contents of arrays $TREE\_VERT$ and $VSET$ for a tree with $N = 6$ randomly generated points.*

TREE_VERT						
i/j	1	2	3	4	5	6
1	1	0	0	0	0	0
2	0	0	0	1	0	0
3	0	1	0	0	0	1
4	0	0	0	0	0	0
5	0	0	1	0	0	0
6	0	1	0	0	0	0

Table 3.1: The TREE_VERT data structure.

VSET		
k	i	j
1	1	1
2	2	4
3	3	2
4	3	6
5	5	3
6	6	2

Table 3.2: The VSET data structure corresponding to table 6.

7. FOR m = 1 TO MAXLOOP
   - FOR $\alpha = 1$ TO N
     - FOR $\beta = 1$ TO N
       - IF $\alpha = \beta$ THEN
         - SET $MTREE[\alpha][\beta](m)$ TO $\sum_{K_\gamma \neq K_\alpha} e^{-m \cdot c(K_\alpha, K_\gamma)} + \delta_1(\alpha)\delta_1(\beta)$
       - ELSE
         - SET $MTREE[\alpha][\beta](m)$ TO $-e^{-m \cdot c(K_\alpha, K_\beta)}$
       - ENDIF
     - ENDFOR
   - ENDFOR
   - Calculate determinant of $MTREE(m)$
   - Take the log of the determinant of $MTREE(m)$
   - Divide log of determinant of $MTREE(m)$ by m.

8. ENDFOR

9. Choose the $\max\{MTREE(m)\}$ to estimate the value of $c(MST)$

10. Print the results.

11. END.

3.3 An Interesting Observation

Remark 3.3.1

$$c(MST) \geq MTREE(m) \qquad \forall m \geq 1$$

Proof. Since,

$$\begin{aligned} \det A(m) &= \sum_{T} \prod_{\alpha,\beta \in T} e^{-m \cdot c(K_\alpha, K_\beta)} \\ &\geq \prod_{\alpha,\beta \in T} e^{-m \cdot c(K_\alpha, K_\beta)} \end{aligned}$$

we have

$$\begin{aligned} \ln \det \mathbf{A(m)} &\geq \sum_{\alpha,\beta \in T} \ln e^{-m \cdot c(K_\alpha, K_\beta)} \\ &= -m \sum_{\alpha,\beta \in T} c(K_\alpha, K_\beta), \end{aligned}$$

which implies,

$$\begin{aligned} \frac{1}{m}\Big[\ln \det \mathbf{A(m)}\Big] &\geq -\sum_{\alpha,\beta \in T} c(K_\alpha, K_\beta) \\ \sum_{\alpha,\beta \in T} c(K_\alpha, K_\beta) &\geq -\frac{1}{m}\Big[\ln \det \mathbf{A(m)}\Big]. \end{aligned}$$

Hence,

$$c(MST) = \min_{T}\Big[\sum_{\alpha,\beta \in T} c(K_\alpha, K_\beta)\Big] \geq -\frac{1}{m}\Big[\ln \det \mathbf{A(m)}\Big] = MTREE(m) \qquad \forall m \geq 1$$

This statement clearly states that the minimum length of the tree will always be greater than or equal to the computed value $MTREE(m)$. Knowing this fact we implement a Kirchoff's Matrix Tree Algorithm called MST_via_MatrixTree_Nu.m. The results of these algorithm are discussed in the next chapter.

Chapter 4

Results

4.1 Results from the AntCol Steiner Algorithm

In order to get sufficiently good results from the limited number of iterations of MAXLOOP, all the parameters were set to the default settings which were taken from the algorithm in the paper by Gurdip Singh, Sanjoy Das, Shekhar V. Gosavi, and Sandeep Pujar [4] since they achieved optimum results with their values. The total length of the tree is calculated to be the total number of dashes in the tree as shown in figure A.4. In the case of figure A.2, the before and after pruning images are displayed. Pruning is the process of removing all non terminal leaves from the resulting Steiner tree.

After each execution of the program, the execution time and tree length was recorded and their averages were calculated over 10 executions of the program with fixed values of N and M where N is the side length of the square grid and M is the number of terminals. These experiments are all preformed on an Intel Pentium 4, 2.6 GHz custom built computer with 1 GB of DDR 400 RAM running Microsoft Windows XP Professional Edition with Service Pack 2. The results obtained are shown in the table 4.1 below. Screen shots and graphical results of AntCol Steiner can be seen in Appendix A. The source code for the entire program can be found in Appendix B to Appendix H.

RESULTS					
M	N	Avg. Steiner Tree Length	Avg. Time (s)	Best Steiner Length	Best Time
9	25	38	0.750	32	0.554
12	30	71	1.062	69	0.996
15	30	73	1.203	71	0.998
20	30	103	1.812	97	1.132
25	30	129	1.907	113	1.787
30	40	170	4.390	167	3.980
40	40	182	4.557	175	4.012
50	40	220	6.915	215	5.788
70	50	340	20.127	305	15.876
100	50	370	25.266	343	17.437

Table 4.1: AntCol Steiner results.

4.2 Results from the Kirchoff's Matrix Tree Algorithm

For the Kirchoff's Matrix Tree theorem we implemented the algorithm via the Matlab programming language. We called our program $MST\_via\_MatrixTree\_Nu(N)$ where N is the dimension of the matrix. In the next section one will find step by step execution of the program. The source code for $MST\_via\_MatrixTree\_Nu(N)$ can be found in Appendix I.

4.2.1 Results from MST_via_MatrixTree_Nu(N) with N = 5

```
This represents the set of random vertices generated by the
program.

VSet =

     1     5     2
     2     1     3
     3     2     3
```

4	5	3
5	2	4

The next table displays the values of MTREE(m) for different values of m. As you can see in this case the value of MTREE(m) when m = 10 is the same as the value of the length of the MST generated by Prim's Algorithm. Also note that we actually approach the length of the MST from below as m got larger.

MinTreeValues =

m	MTREE(m)
1.0000	4.4456
2.0000	5.6934
3.0000	5.9204
4.0000	5.9775
5.0000	5.9933
6.0000	5.9979
7.0000	5.9993
8.0000	5.9998
9.0000	5.9999
10.0000	6.0000

maximum = 6.0000

This show the original distance matrix.

Distance_Matrix =

0	5	4	1	5
5	0	1	4	2
4	1	0	3	1
1	4	3	0	4
5	2	1	4	0

The list below show the edges of the MST. This list is generated using Prim's Algorithm.

MST =

3	2	4	1
5	3	3	4

This length is the exact length of the MST generated by Prim's Algorithm.

MSTree_length = 6

Chapter 5

Conclusions

5.1 AntCol Steiner Conclusion

The ACO Algorithm is a natural algorithm for achieving good estimates of the Rectilinear Steiner Tree Problem. The main achievement here has been to get the algorithm working properly, but it is certainly not written in an optimal way yet and it does not yet compete well with the results claimed in [9]. Upon analyzing the results, my first suggestion for trying to improve the convergence times of the algorithm for large instances of M concerns the pRune method. In the AntCol Steiner program the pRune method carried out a very expensive operation. Thus, in future versions of AntCol Steiner this method will be eliminated. With the current version of the AntCol Steiner program, the Steiner tree has to be pruned because every time an ant is confused it leaves a dead end trail on the graph which is ultimately a non-terminal leaf. An alternative solution to the pRune method would be to use a stack data structure where, the coordinates of each step of the ant are pushed onto the stack until it meets the trail of another ant. If it moves to a vertex and is not confused then we can easily pop the steps back off the stack into its tabu list thereby, marking the ant's trail. On the other hand if the ant is confused it will be placed at a non confusing position within its tabu-list and the ant would restart its walk. Then we would delete all of the coordinates inside the

stack and push the ant's new steps into the stack as soon as the ant starts walking. When the ant finally meets the trail of another ant we will continue as described above. Other improvements to speed up ACO algorithms were suggested in [10]. The improvements included techniques to reduce the search space. ACO-Steiner and AntCol Steiner were implemented on the Hanan's grid, therefore, if one can create a technique to reduce the size of the grid without losing valid information we would in fact reduce the search space. This would cause the algorithm to be much more efficient. The three techniques mentioned in [10] were called Convex Hull Reduction; fulsome Steiner tree (FST) Reduction and Terminal Reduction.

It is our belief, that ACO algorithms are one of best heuristic algorithms used for finding Steiner Trees. This is because they naturally have very short convergent times which makes them very practical.

5.2 Kirchoff's Matrix Tree Conclusion

The results of this experiment verified the theory that the length of a MST can be determined using this theorem. However, there were two main drawbacks to the algorithm. Firstly, the algorithm only had the ability to tell us how long the MST is going to be but, it never produces a graphical representation of the tree. Secondly, the algorithm requires us to calculate the determinant of a matrix called $\mathbf{A(m)}$, which is a more expensive procedure than Prim's algorithm. So in fact we spent more time to get less information. However, originally as mentioned the aim of this approach was to determine the length of a $RSMT$ without actually constructing the tree. The main problem we faced during our experimentation is due to the fact that the Kirchoff's Matrix Tree algorithm cannot be applied directly to the Hanan Grid, because instead of obtaining a $RSMT$ we would obtain a MST with vertices consisting of terminals and possible Steiner points. However, not every vertex in the Hanan Grid will be an actual Steiner point in the $RSMT$. This situation causes our idea to fail since all efforts to maneuver around this problem caused us to loose

the property that $\sum_{k=1}^{N} A_{1k}(m) = 1$ and $\sum_{k=1}^{N} A_{jk}(m) = 0$ from (3.1). Losing this property would cause the Kirchoff's Matrix Tree theorem to become the Kirchoff's Matrix Forest theorem and in this case we would not longer be measuring a single tree.

Bibliography

[1] Beasley, J. E. (1984). An Algorithm for the Steiner Problem in Graphs, Networks, 14: 147-159.

[2] Bonabeau, E. Dorigo, M. Theraulaz, T. (1999). From Natural to Artificial Swarm Intelligence, Oxford University Press, New York.

[3] Chung, F.R.K. and Langlands, R. P. (1996). A Combinatorial Laplacian with vertex weights, Journal of Combinatorial Theory, Series A 75 , no. 2, 316-327.

[4] Das, S., Singh, G., Gosavi, S., and Pujar, S. (2004). Ant Colony Algorithms for Steiner Trees: An Application to Routing in Sensor Networks, Recent Developments in Biologically Inspired Computing, Eds. L. N. de Castro, F. J. von Zuben, Idea Group Publishing, pp. 181-206.

[5] Deneubourg, J. L., Aron, S., Goss, S., and Pasteels, J. M. (1990). The Self-Organizing Exploratory Pattern of the Argentine ant, Journal of Insect Behavior, 3: pp. 159168.

[6] Dorigo, M. and Stützle, T. (2004). Ant Colony Optimization, A Bradford Book, The MIT Press.

[7] Garey, M. R., and Johnson D. S. (1977). The Rectilinear Steiner Tree Problem is NP-Complete. SIAM Journal of Applied Mathematics, 32, pp. 855 - 859.

[8] Hanan, M. (1966). On Steiner's problem with rectilinear distance, SIAM Journal of Applied Math, 14(2), pp. 255-265.

[9] Hu, Y., Jing, T., Hong, X., Feng, Z., Hu, X. and Yan, G. (2005). ACO-Steiner: Ant Colony Optimization Based Efficient Rectilinear Steiner Minimal Tree Construction, http://www.paper.edu.cn.

[10] Hu, Y., Jing, T., Hong, X., Feng, Z., Hu, X. and Yan, G. (2006). ACO-Steiner: Ant Colony Optimization Based Rectilinear Steiner Minimal Tree Algorithm, J. Comput. Sci. and Technol. Vol 21, No. 1, pp. 147-152.

[11] Kaltofen, E. and Villard, G. (2001) On the Complexity of Computing Determinants, Proc. Fifth Asian Symposium on Computer Mathematics ASCM 2001 at Ehime University, Japan Sep 26-28, World Scientific Publishing Company, Singapore.

[12] Kirchhoff, G. (1847). Ann. Phys. Chem. 72, 497; Brooks, R.L., Smith, C.A.B., Stone, A.H. and Tutte, W.T. (1940). Duke Math. J. 7, 312; Nerode, A. and Shank, H. (1961). Amer. Math. Monthly 68, 244.

[13] Prim, R. C. (1957). Shortest connection networks and some generalisations. Bell System Technical Journal, 36, pp. 1389 – 1401.

[14] Warme, D., Winter, P., and Zachariasen, M. (2001). GeoSteiner 3.1 http://www.diku.dk/geosteiner .

Appendix A

Graphical Results Obtain from AntCol Steiner

In this section we display the user screens and program results in a graphical form. The tree in figure A.4 contains $M = 50$ terminals and is created using the Hanan grid A.3 as its initial graph. The input graphical user interface consists of three display panels namely Parameters, Inputs and Outputs. In the Parameters panel a list of all of the program's parameters are displayed with various values to allow the user the experiment with the parameter values. The recommended values for the program are the default values which can be selected by clicking the $Edit -> Default\ Settings$ or by pressing F2. The Help Manual can be accessed by clicking $Help -> Manual$ or by pressing F1. Selecting $rho = 1.0$ produces **NO** tree as it evaporates the entire pheromone trail. The Inputs panel consisted of two input fields. The first input field N is designed for the user to enter the size of the underline grid they wished to work with. While the second input field M is used to allow the user to specify how many random terminals to generate. Finally, the Outputs panel is used to display the time taken in milliseconds to execute the program and the length of the tree generated. In both the Hanan Grid A.3, and the Steiner Tree A.4 each vertical or horizontal dash has a length of one unit while each vertex had a length of zero.

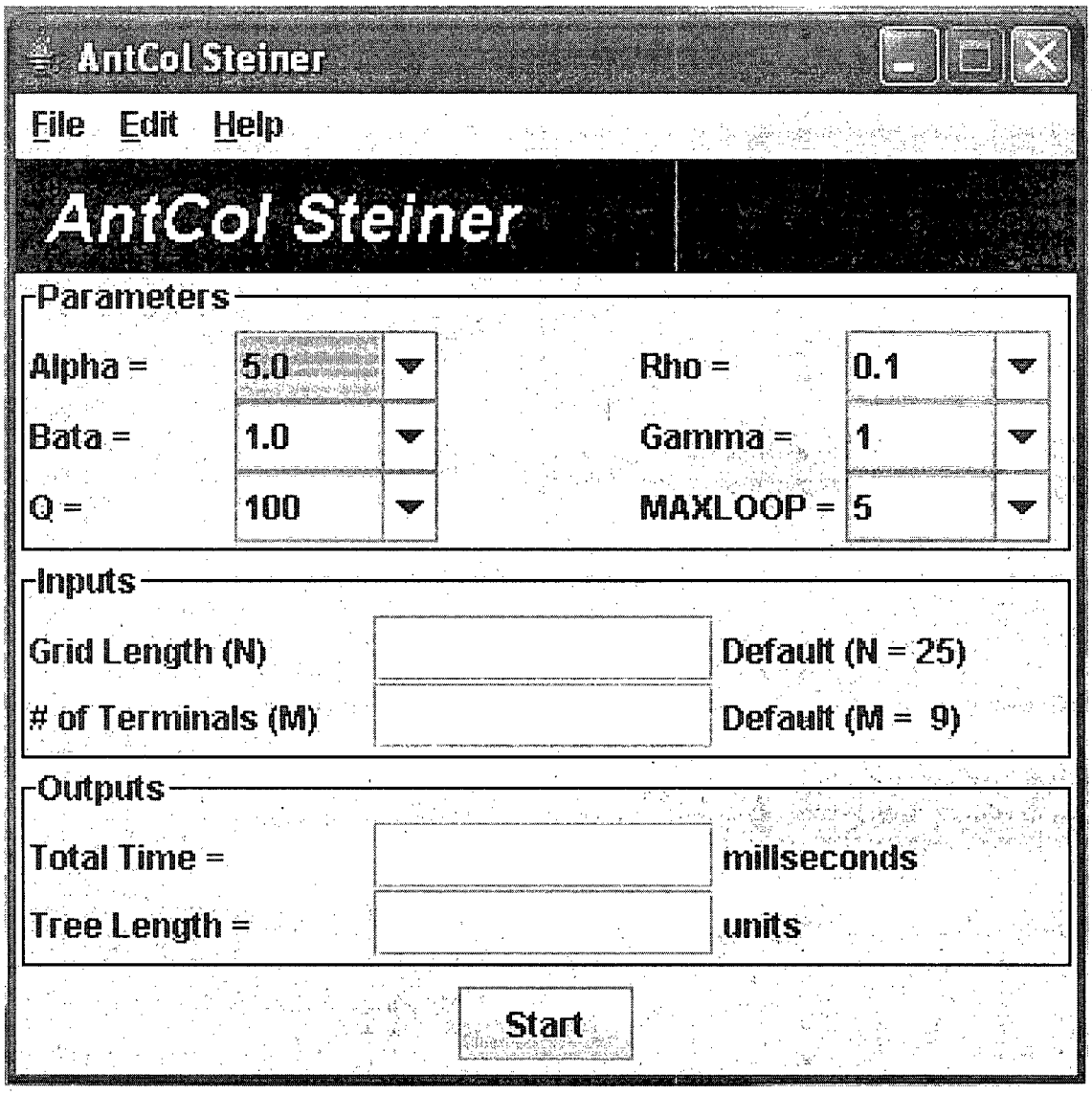


Figure A.1: The Graphical User Interface of the AntCol Steiner Program.

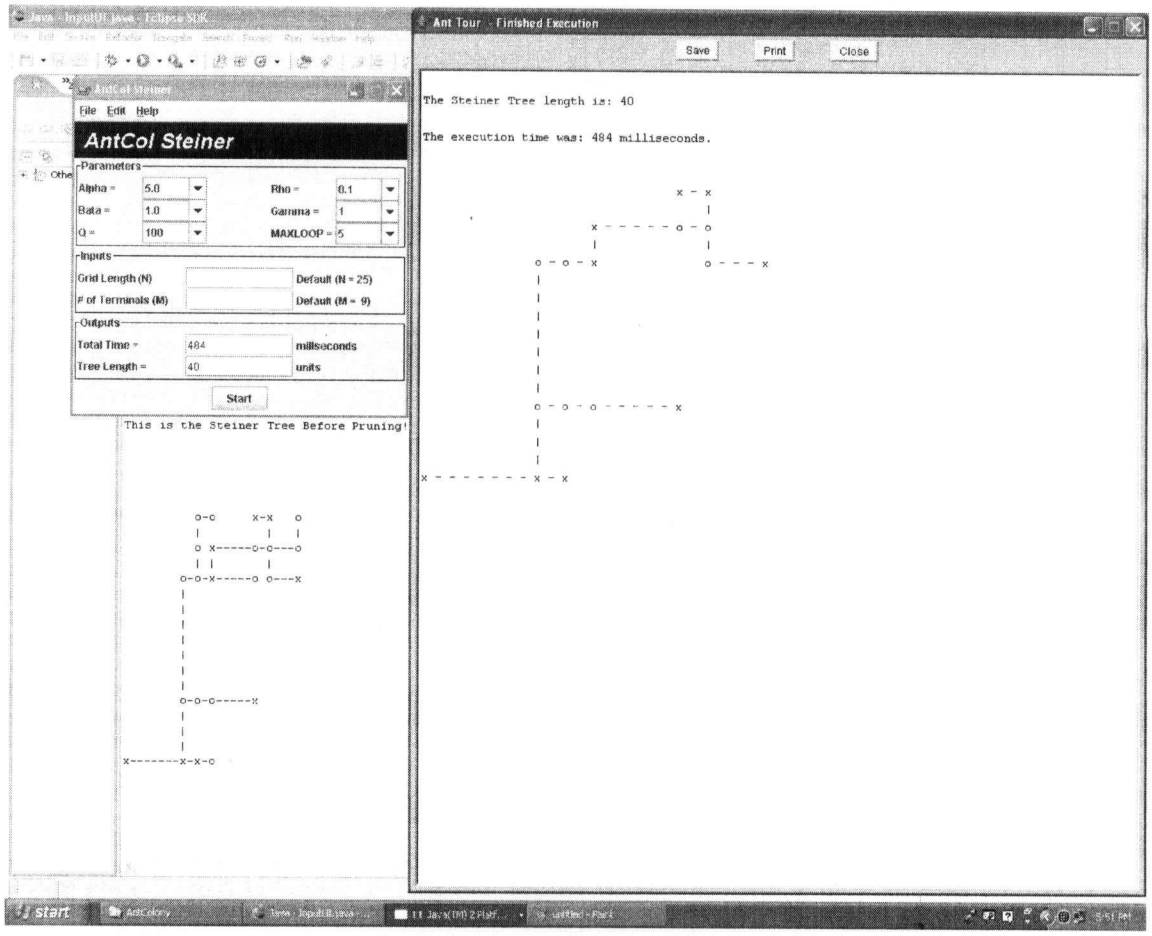


Figure A.2: This Screen show the Steiner Tree before and after Pruning. The pre-pruned tree is located in the bottom left hand corner of the figure. The pruned tree is on the right hand side of the figure.

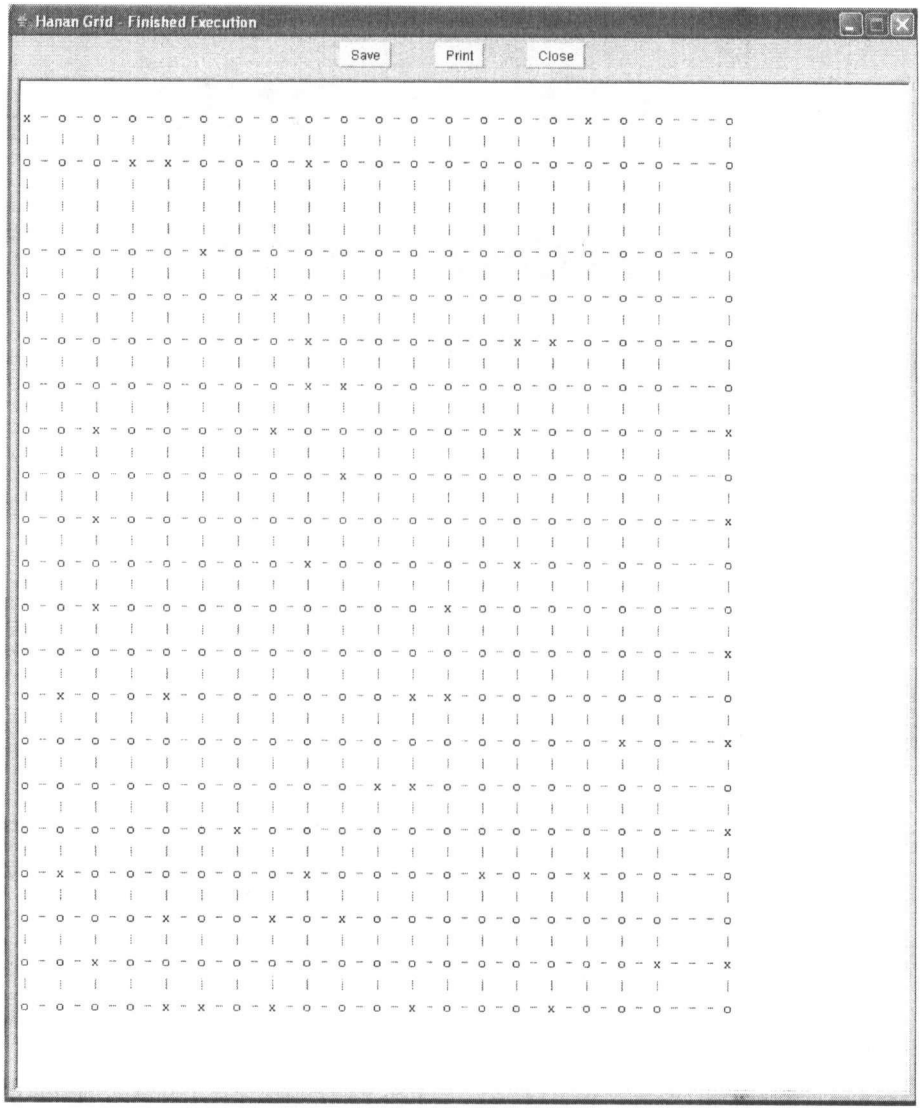


Figure A.3: Hanan Grid with M = 50 terminals which was generated using AntCol Steiner. On this grid the "x" represents the terminal vertices and the "o" represent the possible Steiner points.

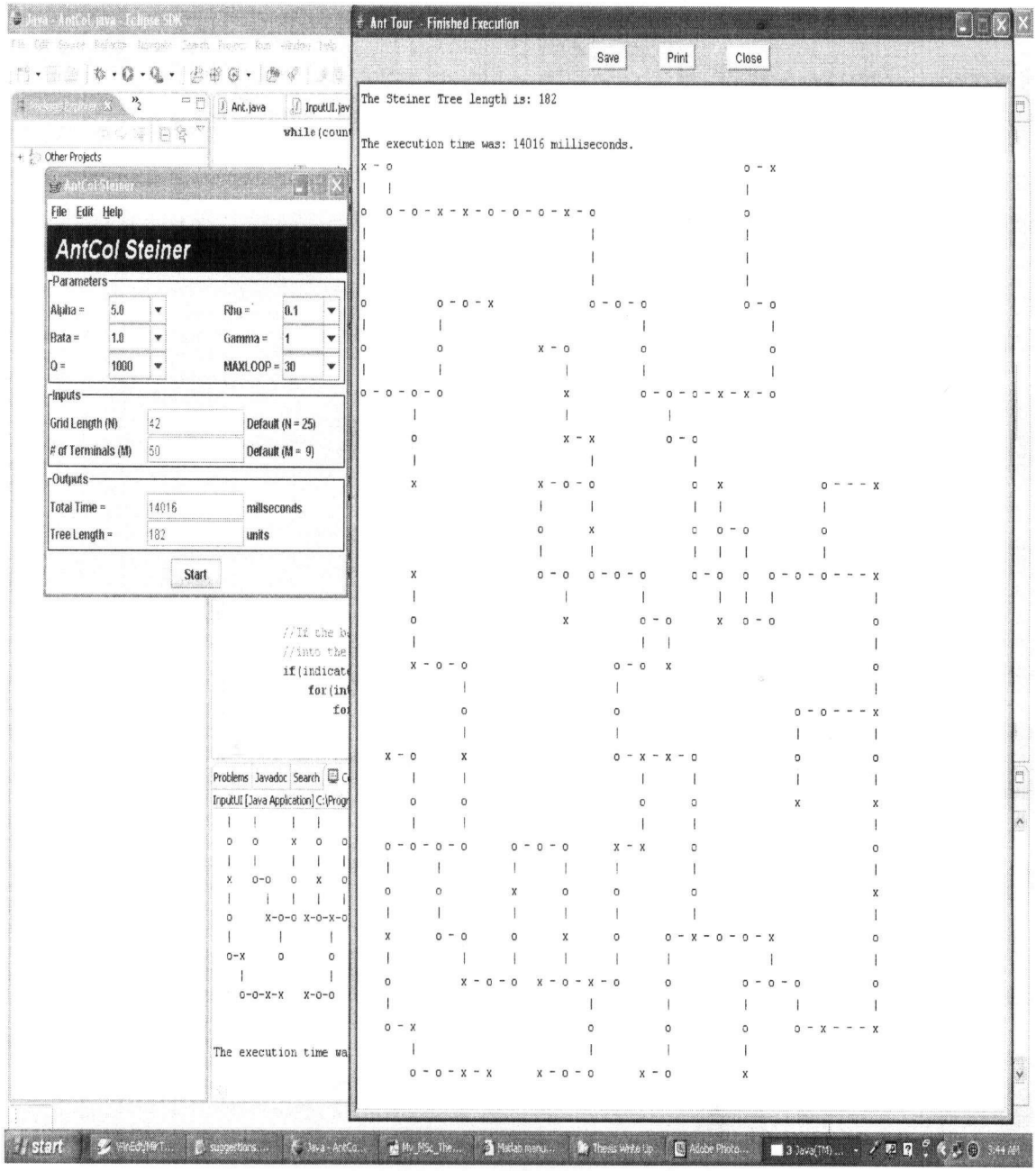


Figure A.4: The Steiner Tree with M = 50 terminals; Tree Length = 182 units and Total Time = 14016 milliseconds obtained after using the AntCol Steiner program on the Hanan Grid above A.3. On this tree the "x" represents the terminal vertices and the "o" represent the Steiner points. The parameter values are Alpha = 5, Beta = 1, Q = 1000, Rho = 0.1, Gamma = 1 and MAXLOOP = 30.

Remark A.0.1 *The solution presented in figure A.4 is not an optimal solution.*

Appendix B

Source Code for InputUI.java

Remark B.0.2 *This class contains the Java main method. The other methods in this class are designed to generate, display and respond to all of the graphical user interface components presented in figure A.1. This class does not contain any ACO algorithm properties.*

```
package AntSteiner.ui;
import javax.swing.*;
import AntSteiner.*;
import java.awt.*;
import java.awt.event.*;

/**
 * Description: This class handles all of the imput GUI stuff.
 * Class: InputUI.java
 * @author Anthony Blackman
 * @version April 10, 2006
 */
public class InputUI extends JFrame implements ActionListener{
```

```
/**This variable was used as a reference to the AntCol class*/
private static AntCol ac;

/**This variable was used to store the number of terminals
generated*/
private static JTextField M;

/**This variable was used to store the dimension of the
generated*/
private static JTextField N;

/**This variable was used to store M and N*/

private static int [] gridLTermN;

/**This is a reference to the JTextField*/

private static JTextField timeTaken;

/**This is a reference to the JTextField*/

private static JTextField steinerLength;

/**This is a reference to the JLabel*/

private JLabel defaultN;

/**This is a reference to the JLabel*/
```

```
private JLabel defaultM;

/**This is a reference to the Start button*/

private JButton run;

/**This is a reference to the JComboBox*/

private static JComboBox alphaBox;

/**This is a reference to the JComboBox*/

private static JComboBox bataBox;

/**This is a reference to the JComboBox*/

private static JComboBox rhoBox;

/**This is a reference to the JComboBox*/

private static JComboBox maxloopBox;

/**This is a reference to the JComboBox*/

private static JComboBox gammaBox;

/**This is a reference to the JComboBox*/
```

```
private static JComboBox qBox;

/**This is a reference to the parameter values of the program
 * the values include alpha, bata, gamma, rho, Q, maxloop*/
public static String [] parameters;

/**This is a reference to the JFrame*/

private JFrame f;

/**This is a reference to the InputUI class*/

private static InputUI ui;

/**This is the default constructor.*/
public InputUI(){
        gridLTermN = new int [2];
        inputScreen();
    }
/**
 * Description: This method creates the input user interface
 *(AntCol Steiner) screen.
 */
public int [] inputScreen(){
    f = new JFrame("AntCol Steiner");
    f.getContentPane();
    JMenuBar mainMenuBar = new JMenuBar();
```

```
    //This creates a file Menu with a mnemonic under the F.
    JMenu fileMenu = new JMenu("File");
    mainMenuBar.add(fileMenu);
    fileMenu.setMnemonic(KeyEvent.VK_F);

    //This creates a edit Menu with a mnemonic under the E.
    JMenu editMenu = new JMenu("Edit");
    mainMenuBar.add(editMenu);
    editMenu.setMnemonic(KeyEvent.VK_E);

    //This creates a help Menu with a mnemonic under the H.
    JMenu helpMenu = new JMenu("Help");
    mainMenuBar.add(helpMenu);
    helpMenu.setMnemonic(KeyEvent.VK_H);

    //This creates a clear menu item under the Edit menu.
    JMenuItem clear = new JMenuItem("Clear", KeyEvent.VK_C);
    editMenu.add(clear);

    //This creates a default menu item under the Edit menu.
JMenuItem defaultSettings = new JMenuItem("Default Settings",
KeyEvent.VK_D);
    editMenu.add(defaultSettings);

    //This creates an exit mnemonic under the exit menu item in the file menu.
    JMenuItem exit = new JMenuItem("Exit",KeyEvent.VK_X);
    fileMenu.add(exit);
```

```
    //This creates an instruction manual.
    JMenuItem programHelp = new JMenuItem("Manual", KeyEvent.VK_M);
    helpMenu.add(programHelp);

    //This creates an about mnemonic under the about
    //menu item in the help menu.
    JMenuItem about = new JMenuItem("About",KeyEvent.VK_A);
    helpMenu.add(about);

    //This also allow the about dialog box to pop up
    //when the F3 key is pressed.
    about.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_F3, 0, false));

    //This is the action listener that pops up the dialog box
    //and exit when the exit button is pressed.
    exit.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            JOptionPane exitFrame = new JOptionPane();
JOptionPane.showMessageDialog(exitFrame,"Good Bye", "Exit AntCol
Steiner",
                    JOptionPane.INFORMATION_MESSAGE);
            System.exit(0);
        }
    });

    //This is the action listener that pops up the about
    //dialog box and displays the about information
```

```
//when the about menu button is pressed.
about.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        JOptionPane aboutFrame = new JOptionPane();

        JOptionPane.showMessageDialog(aboutFrame,
"AntCol Steiner\nVersion 2.0\nBy Anthony Blackman", "About AntCol
 Steiner",
                JOptionPane.INFORMATION_MESSAGE);
        }
});
//This clears all text fields.
clear.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        N.setText("");
        M.setText("");
        timeTaken.setText("");
        steinerLength.setText("");
    }
});

//This clears all text fields.
defaultSettings.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        alphaBox.setSelectedIndex(4);
        bataBox.setSelectedIndex(0);
        rhoBox.setSelectedIndex(1);
        maxloopBox.setSelectedIndex(4);
```

```
            gammaBox.setSelectedIndex(0);
            qBox.setSelectedIndex(0);
            N.setText("");
            M.setText("");
            timeTaken.setText("");
            steinerLength.setText("");
        }
    });

    //This sets the defaultSettings when the F2 key is pressed.
    defaultSettings.setAccelerator(KeyStroke.getKeyStroke
    (KeyEvent.VK_F2, 0, false));
    //This displays the help manual when F1 is pressed.
    programHelp.setAccelerator(KeyStroke.getKeyStroke
    (KeyEvent.VK_F1, 0, false));
    //This displays the help manual all text fields.
    programHelp.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            JFrame textFrame = new JFrame("AntCol Steiner Help");
            JPanel textPanel = new JPanel();
            JLabel textTitle = new JLabel("Help Manual");
            textPanel.setLayout(new BorderLayout());
            JTextArea ta = new JTextArea();
ta.setText("\nFirst select the parameter values from the parameter panel.\n" +
"The default values are recommended for the best performance. Then\n" +
"click the Start button and the program will run with the currently \n" +
"selected parameter values and the grid size and terminal number unless\n" +
"otherwise specified. The default grid size and terminal numbers are N = 25\n" +
```

```
"and M = 9 respectively. This values produces a N x N  (25 x 25) grid \n" +
"with M = 9 terminals. The user can also enter their preferred values \n" +
"of N and M. However, the value of M enter must always be less than N \n" +
"squared.\n\nThe Edit -> Clear menu button, clears all input and output \n" +
"fields.\n\nThe Constants panel " +
"displays all of the constant parameter \nvalues used to
perform the Ant Colony Optimization " +
"on the \nHanan's Grid\n\n\nAuthor: Anthony Blackman");
ta.setSize(500,500);
JScrollPane helpScroll = new JScrollPane
(ta,ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
ScrollPaneConstants.HORIZONTAL_SCROLLBAR_ALWAYS );
            //just getting a little fancy...
            Font font = new Font("Arial Bold", Font.ITALIC, 18);
            textTitle.setFont(font);
            textTitle.setOpaque(true);
            textTitle.setForeground(Color.WHITE);
            textTitle.setBackground(Color.BLUE.darker());
            textTitle.setPreferredSize(new Dimension(400, 24));
            textTitle.setBorder(BorderFactory.createEmptyBorder(0, 10, 0, 10));
            //Add TextTitle and HelpScroll to TextPanel.
            textPanel.add(textTitle, BorderLayout.NORTH);
            textPanel.add(helpScroll, BorderLayout.CENTER);
            //Add TextTitle to the Frame.
            textFrame.add(textPanel, BorderLayout.CENTER);
            textFrame.setSize(330,330);
            textFrame.setLocationRelativeTo(null);
            textFrame.setVisible(true);
```

```
    }
});

// This adds the main menu bar to the panel
f.setJMenuBar(mainMenuBar);

JPanel title = new JPanel();

JLabel programTitle = new JLabel("AntCol Steiner");
title.setLayout(new BorderLayout());
// just getting a little fancy...
Font font = new Font("Arial Bold", Font.ITALIC, 24);
programTitle.setFont(font);
programTitle.setOpaque(true);
programTitle.setForeground(Color.WHITE);
programTitle.setBackground(Color.BLUE.darker());
programTitle.setPreferredSize(new Dimension(400, 40));
programTitle.setBorder(BorderFactory.createEmptyBorder(0, 10, 0, 10));
// create the course title bar
title.add(programTitle, BorderLayout.CENTER);

JPanel constantsPanel = new JPanel();
constantsPanel.setLayout(new GridLayout(3,5));

JLabel alpha = new JLabel("Alpha = ");
JLabel rho = new JLabel ("Rho = ");
JLabel bata = new JLabel ("Bata = ");
```

```
JLabel gamma = new JLabel ("Gamma = ");
JLabel Q = new JLabel ("Q = ");
JLabel maxLoop = new JLabel ("MAXLOOP = ");
JLabel space1 = new JLabel("  ");
JLabel space2 = new JLabel("  ");
JLabel space3 = new JLabel("  ");

String [] alphaValues = {"1.0","2.0","3.0","4.0","5.0" };
String [] bataValues = {"1.0","2.0","3.0","4.0","5.0" };
String [] gammaValues = {"1","2","3","4","5" };
String [] maxloopValues = {"1","2","3","4","5","6","7","8",
                           "9","10","20", "30", "40", "50" };
String [] qValues = {"100","1000","10000" };
String [] rhoValues = {"0.0","0.1","0.2","0.3","0.4","0.5",
                       "0.6","0.7","0.8","0.9","1.0"};

alphaBox = new JComboBox(alphaValues);
bataBox = new JComboBox(bataValues);
rhoBox = new JComboBox(rhoValues);
maxloopBox = new JComboBox(maxloopValues);
gammaBox = new JComboBox(gammaValues) ;
qBox = new JComboBox(qValues);

alphaBox.setBackground(Color.WHITE);
bataBox.setBackground(Color.WHITE);
rhoBox.setBackground(Color.WHITE);
maxloopBox.setBackground(Color.WHITE);
gammaBox.setBackground(Color.WHITE);
```

```
qBox.setBackground(Color.WHITE);

alphaBox.setSelectedIndex(4);
bataBox.setSelectedIndex(0);
rhoBox.setSelectedIndex(1);
maxloopBox.setSelectedIndex(4);
gammaBox.setSelectedIndex(0);
qBox.setSelectedIndex(0);

constantsPanel.add(alpha);
constantsPanel.add(alphaBox);
constantsPanel.add(space1);
constantsPanel.add(rho);
constantsPanel.add(rhoBox);
constantsPanel.add(bata);
constantsPanel.add(bataBox);
constantsPanel.add(space2);
constantsPanel.add(gamma);
constantsPanel.add(gammaBox);
constantsPanel.add(Q);
constantsPanel.add(qBox);
constantsPanel.add(space3);
constantsPanel.add(maxLoop);
constantsPanel.add(maxloopBox);
//This puts a Border around the constants panel with
//the words Marks on the border.
constantsPanel.setBorder( BorderFactory. createTitledBorder
( BorderFactory. createLineBorder( Color.BLACK), "Parameters"));
```

```
JPanel mainPanel = new JPanel();

JPanel inputPanel = new JPanel();
JPanel outputPanel = new JPanel();
inputPanel.setLayout(new GridLayout(2,3));
outputPanel.setLayout(new GridLayout(2,3));

mainPanel.setLayout(new BorderLayout());
mainPanel.add(constantsPanel, BorderLayout.NORTH);

JLabel gridSize = new JLabel ("Grid Length (N) ");
N = new JTextField (15);
defaultN = new JLabel(" Default (N = 25)");
defaultM = new JLabel (" Default (M =  9)");
JLabel terminalNum = new JLabel ("# of Terminals (M)");
M = new JTextField (15);
JLabel time = new JLabel ("Total Time = ");
timeTaken = new JTextField (10);
JLabel length = new JLabel("Tree Length = ");
steinerLength = new JTextField (10);
JLabel ms = new JLabel(" millseconds");
JLabel units = new JLabel (" units");

inputPanel.add(gridSize);
inputPanel.add(N);
inputPanel.add(defaultN);
inputPanel.add(terminalNum);
```

```
inputPanel.add(M);
inputPanel.add(defaultM);
inputPanel.setBorder( BorderFactory. createTitledBorder
( BorderFactory. createLineBorder( Color.BLACK), "Inputs"));
outputPanel.add(time);
outputPanel.add(timeTaken);
outputPanel.add(ms);
outputPanel.add(length);
outputPanel.add(steinerLength);
outputPanel.add(units);
outputPanel.setBorder( BorderFactory. createTitledBorder
( BorderFactory. createLineBorder( Color.BLACK), "Outputs"));

JPanel iOPanel = new JPanel();
iOPanel.setLayout(new GridLayout(2,1));

iOPanel.add(inputPanel);
iOPanel.add(outputPanel);

run = new JButton("Start");

JPanel buttonPanel = new JPanel();
buttonPanel.add(run);

mainPanel.add(iOPanel, BorderLayout.CENTER);

f.add(title, BorderLayout.NORTH);
f.add(mainPanel, BorderLayout.CENTER);
```

```
    f.add(buttonPanel, BorderLayout.SOUTH);
    f.setSize(380,380);
    f.setResizable(false);
    f.setVisible(true);
    //f.setLocationRelativeTo(null);
    //This puts the AntCol Steiner screen in the center.
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    return gridLTermN;
} /**
 *Description: This methods looks for the run button
 *to be clicked and creates a seperate thread
 *for the execution of the program.
 */
public void actionPerformed(ActionEvent e) {
    Steiner s = new Steiner();
    Thread runAnt = new Thread(s);
    runAnt.start();

}

/**
 * Description: This methods extracts the information
 * from the text fields if values were entered
 * else the method would create default values
 * to be used with in the program.
 */
public static void extract(){
```

```
    ac = new AntCol();
    ac.setTook(0);
    try {
        gridLTermN [0] = Integer.parseInt(N.getText());
        //This converts the string from the Text Field
        //into an integer.
        gridLTermN [1] = Integer.parseInt(M.getText());
        //This converts the string from the Text Field
        //into an integer.
     }
    catch (NumberFormatException nfe1){
            gridLTermN [0] = 25;
            gridLTermN [1] = 9;
        }
    parameters = new String[6];
    parameters[0] = (String)alphaBox.getSelectedItem();
    parameters[1] = (String)bataBox.getSelectedItem();
    parameters[2] = (String)rhoBox.getSelectedItem();
    parameters[3] = (String)maxloopBox.getSelectedItem();
    parameters[4] = (String)gammaBox.getSelectedItem();
    parameters[5] = (String)qBox.getSelectedItem();

    ac.test(gridLTermN [0], gridLTermN [1]);
    timeTaken.setText(ac.getTook() + "");
    steinerLength.setText(ac.getBestSlength() + "");

} /**
 * @param args
```

```
 */
public static void main(String[] args) {
    // TODO Auto-generated method stub
    ui = new InputUI();
    ui.run.addActionListener(ui);
  }
} /**
 * This class implements the Runnable Interface. This makes
 * the stuff in the run method excute using a separte thread.
 * This thread was used because sometimes the excution time of
 * the program can be very long and using a seperate thread
 * allow the user access to the GUI controls while the programs
 * is executing.
 */
class Steiner implements Runnable{
    /**
     * This is the overloaded run method associated with the thread.
     */
    public void run(){
           {
               try {
                   InputUI.extract();
                   Thread.sleep(5);
                  }
               catch(InterruptedException e) {}}}}
```

Appendix C

Source Code for AntCol.java

```
package AntSteiner; import AntSteiner.util.*; import
AntSteiner.ui.*;

/**
 * Class: AntCol.java
 * @author Anthony Blackman
 * @version April 10, 2006
 */
public class AntCol {

/**This data structure store the list of the terminals
 * in the Hanan's Grid*/
private int [][] terminals;

/**This data structure was used to store the Hanan's grid.*/
private String [][] grid;

/**This data sturture was used to store the current Steiner
```

```
tree.*/

private String [][] steinerTree;

/**This data sturture was used to store the best Steiner tree.*/
private String [][] bestSteinerTree;

/**This data sturture was used to store intensity levels of the
 * edges of the Hanan's Grid.*/
private double [][] intensity;

/**This variable was used to store the number of iterations of the
system.*/

private int maxLoop;

/**This variable was used to store the quality of the trail*/
private int Q;

/**This variable was used to store the length of the current
steiner tree.*/
private int sLength = 0;

/**This variable was used to store the length of the best steiner
tree*/

private int bestSlength = 0;
```

```
/**This variable was used to determine whether or not
 * a better Steiner tree was found*/
private int indicate;

/**This variable was used to record the start time.*/

private long start;

/**This variable was used to record the finish time.*/

private long end;

/**This variable was used to record the time taken to execute the
program*/
private long took;

/**This variable was used to determine the levels of phermone
evaporation*/

private double rho;

/**This variable was used to reference the input display*/

private Display d;

/**This variable was used as a reference to the Hanan Grid.*/
private HananGrid hg;
```

```
/**This variable was used as a reference to the TreeConstruct
class.*/

private TreeConstruct tree;

/**This data structure was used to store the boundary values of
the Hanan Grid.*/

private int [] gridLimits;

    public AntCol(){

        }

    public void test(int n, int m){

        rho = Double.parseDouble(InputUI.parameters[2]);
        maxLoop = Integer.parseInt(InputUI.parameters[3]);
        Q = Integer.parseInt(InputUI.parameters[5]);

        int count = 0; // This is the number of loops.

        double pheromone = 1000;
        gridLimits = new int [4];
        steinerTree = new String[n][n];
        bestSteinerTree = new String [n][n];

        grid = new String[n][n];
```

```
intensity = new double[n][n];
terminals = new int [m][2];
hg = new HananGrid();
tree = new TreeConstruct();
d = new Display();

start = System.currentTimeMillis(); // Start the clock.


grid = hg.gridGenerator(n, m, gridLimits);
//This generates the Hanan grid.
intensity = hg.initial_intensity(grid, pheromone);
//This array contains the trail intensity.
terminals = hg.getPoints();
// This array contains the terminal points.

//This is the first steiner tree it will be used as
//a test standard to compare the other trees.
bestSteinerTree = tree.constructSteinerTreeByACO
(terminals, grid, intensity, gridLimits);
bestSlength = tree.getTLength();

// Here at the other steiner trees.
while(count < maxLoop){

//These loops updates the trail intensity for all of the paths that
where crossed.
for(int row = 0; row < steinerTree.length; row++){
```

```
for(int col = 0; col < steinerTree.length; col++) {
    if(bestSteinerTree[row][col]== "*" || bestSteinerTree[row][col]== "a" ){
    //This increases the level of pheromones.
    intensity[row][col] = updateIntensity(sLength, row, col, intensity);
    //This updates the edges
    //which have been crossed.
            }
            else
            {

     intensity[row][col] = ((double)1 - rho)*intensity[row][col];
            }
        }
      }

steinerTree = tree.constructSteinerTreeByACO(terminals, grid,
intensity, gridLimits);
      sLength = tree.getTLength();

      //This indicates if the firstSlength is less than sLength.
      //if indicate = 1 firstSlength is shorter than sLength.
      //if indicate = 0 firstSlength is longer
      //than sLength. If indicate = -1 the test failed.
      indicate = chooseBestSol(bestSlength, sLength);

      //If the bestSteinerTree is longer we copy the better tree
      //(ie. steinerTree) into the bestSteinerTree. Then get its length.
  if(indicate == 0){
```

```
    for(int row = 0; row < steinerTree.length; row++){
       for(int col = 0; col < steinerTree.length; col++) {
         bestSteinerTree[row][col]= steinerTree[row][col];
       }
    }
 System.out.println("\n\nA better Steiner Tree was found!");
 bestSlength = sLength; //set the bestSlength to be the shorter distance.
 }
  else if (indicate == -1)
        {
  System.out.println("The comparsion test failed!");
        }

        count++; // This increments the MAXLOOP
     } // This is the end of the MAXLOOP.
        end = System.currentTimeMillis(); //Stop the clock

        took = end - start; // Elapsed time.
        d.displayTree(bestSteinerTree, grid, bestSlength, took);

System.out.println ("\nThe execution time was: " + took + "
milliseconds. ");



    }

/**
* Description: This method compares two numbers and then returns an
```

```
* indicator that indicates which number is the bigger. if indicate = 1
* firstSlength is shorter than sLength. if indicate = 0 firstSlength is longer
* than sLength. If indicate = -1 the test failed.
*
* @param bestSlength The shortes steiner tree that was generated.
* @param sLength The current steiner tree which was just generated.
* @return indicator This was used to indicate whether or not
* the current steiner tree
* is better than the current best steiner tree.
*/
    public int chooseBestSol(int bestSlength, int sLength){
        int indicator = -1;

        if(bestSlength < sLength){
            indicator = 1;
        }
        else
        {
            indicator = 0;
        }
        return indicator;
    }

    /**
     * Description: This method was used to update the trail intensity.
     * @param sLength This is the length of the steiner tree.
     * @param row This is a x coorinate of a point on the steiner tree.
     * @param col This is a y coorinate of a point on the steiner tree
```

```
 * @param intensity This is the intensity matrix.
 * @return intensityValue This the value of the intensity at
 * that point on the trail.
 *
 * */
public double updateIntensity(int sLength, int row, int col,
double [][] intensity){
    double intensityValue = 0.0;
    double delta_tau;
    if (sLength == 0.0)
    {
        delta_tau = 0.0;
    }
    else
    {
        delta_tau =  Q / sLength;
    }

    intensityValue = (1 - rho)*intensity[row][col] + rho * delta_tau;
    //This is the trail intensity.

    return intensityValue;
}

/**
 * @return Returns the intensity.
 */
public double[][] getIntensity() {
```

```
    return intensity;
}

/**
 * @param intensity The intensity to set.
 */
public void setIntensity(double[][] intensity) {
    this.intensity = intensity;
}

/**
 * @return Returns the bestSlength.
 */
public int getBestSlength() {
    return bestSlength;
}

/**
 * @return Returns the took.
 */
public long getTook() {
    return took;
}

/**
 * @param took The took to set.
 */
public void setTook(long took) {
```

```
        this.took = took;
    }
}
```

Appendix D

Source Code for HananGrid.java

```
package AntSteiner.util; import java.util.*; import AntSteiner.ui.*;


/**
 * Class: HananGrid.java
 * @author Anthony Blackman
 * @version April 10, 2006
 */
public class HananGrid {

/**This data structure was used to store the Hanan's Grid*/

private String [][] grid;

/**This data structure was used to store the trail intensities.*/
private double [][] intensity;

/**This reference was used to access the input user interface.*/
```

```
private Display d;

/**This data structure was used to store the points terminals of the
system.*/
private int [][] points;

/**This variable was used to store the x coordinate of the Hanan's
grid*/

private int x;

/**This variable was used to store the y cooridinate of the Hanan's
grid*/

private int y;

/**This variable was used to store the smallest x cooridinate of the
Hanan's grid*/

private int xmin;
/**This variable was used to store the largest x
cooridinate of the Hanan's grid*/

private int xmax;

/**This variable was used to store the smallest y cooridinate of the
Hanan's grid*/
```

```
private int ymin;
/**This variable was used to store the largest y
cooridinate of the Hanan's grid*/

private int ymax;

/**This variable was used to store the reference
to the random number generator*/

private Random ran;

/**HananGrid Constructor*/ public HananGrid(){
    x = 0;
    y = 0;
    xmin = 0;
    xmax = 0;
    ymin = 0;
    ymax = 0;

} /**This method was design to generate a Hanan Grid from a random
 * set of n integer vertices. The "x" represents the terminal points
 * The "o" represents the possible steiner points. The "|"
 * and "-" represents the edges of the grid and the " " are places
 * that the ant cant walk
 * @param n
 * @return grid Which is a representation of the Hanan' Grid.
 */
```

```
public String [][] gridGenerator(int n, int m, int [] gridLimits){
    grid = new String[n][n]; //Size of the grid is n by n.
    points = new int[m][2];
    d = new Display();
    ran = new Random(); // Creates a new random number generator.


    for (int i = 0; i < grid.length; i++) {
        for(int j = 0; j < grid.length; j++){
            grid[i][j] = " "; // Initilize the grid array with blank spaces.
        }
    }

    for (int i = 0; i < points.length; i++){
        x = ran.nextInt(n); //generates a random number between 0 and n.
        y = ran.nextInt(n); //generates a random number between 0 and n.
        x = 2*x; // This makes sure that the x coordinates is always even
        y = 2*y; // This makes sure that the y coordinates is always even

        if (x < grid.length && y < grid.length){
            for(int k = 0; k < grid.length; k++){
                if(grid[x][k]== "|" || grid[x][k]== "o" ){
                    grid[x][k] = "o";
                    }
                else {
                    grid[x][k]= "-";
                }
```

```
            if(grid[k][y]== "-" || grid[k][y]== "o" ){
                grid[k][y] = "o";
            }
            else {
            grid[k][y] = "|";
            }
        }

        points[i][0] = x; // This stores the x points
        points[i][1] = y; //This stores the y points.
    }
    else
    {
        i--;
    }
}

//This for loop was used to look for the point just generated and plac
//an X at the point.
for (int i = 0; i < points.length; i++) {
    grid[points[i][0]][points[i][1]]= "x";
}
xmin = grid.length;
ymin = grid.length;

for(int s = 0; s < points.length; s++){
    if(points[s][0] <= xmin )   {
```

```
        xmin = points[s][0]; // This find the lowest x point.
    }
    if(points[s][1] <= ymin )   {
        ymin = points[s][1];// This find the lowest y point.
    }
    if (xmax <= points[s][0]  ) {
        xmax = points[s][0];// This find the highest x point.
    }
    if(ymax <= points[s][1] )   {
        ymax = points[s][1]; // This find the higest y point.
    }
}
//Trims off excess grid from the top.
if (xmin > 0){
    for(int k = 0; k < xmin; k++ ){
        for(int i = 0; i < grid.length; i++){
            grid[k][i]= " ";
        }
    }
}
//This trim excess of the grid on the left side.
if (ymin > 0){
    for(int k = 0; k < ymin; k++ ){
        for(int i = 0; i < grid.length; i++){
            grid[i][k]= " ";
        }
    }
}
```

```
    //This trim excess of the grid from the right side.
    if (ymax < grid.length){
        for(int k = ymax + 1; k < grid.length; k++ ){
            for(int i = 0; i < grid.length; i++){
                grid[i][k]= " ";
            }
        }
    }

//  Trims off excess grid from the bottom.
    if (xmax < grid.length){
        for(int k = xmax + 1; k < grid.length; k++ ){
            for(int i = 0; i < grid.length; i++){
                grid[k][i]= " ";
            }
        }
    }

    gridLimits[0] = xmax;
    gridLimits[1] = ymax;
    gridLimits[2] = xmin;
    gridLimits[3] = ymin;

    d.displayGrid(grid); // This method call displays the Hanan's Grid.


    return grid; //This returns the Hanan's Grid.
```

```
}

/**
 * Discription: The string [][] is a copy of the Hanan grid, the double
 * parameter is for the initial intensity of the trail and the double
 * array is a intensity matrix. If the there is an edge in the Hanan grid
 * then the intensity is set to the initial_intensity. If there is no edge
 * at position i, j in the Hanan grid then the intensity is set to 0.0. After
 * all edges have been check we return the initial edge intensity matrix.
 *
 * @param grid
 * @param initial_intensity
 * @return intensity
 *
 * */
public double [][] initial_intensity(String [][] grid, double
initial_intensity ){
    int n = grid.length; //Dimensions of the Hanans grid.

    intensity = new double [n][n];

    for (int i = 0; i < intensity.length; i++){
        for (int j = 0; j < intensity.length; j++){
            if(grid[i][j]== " "){
                intensity[i][j]= 0.0;
            }
            else {
```

```
                intensity[i][j] = initial_intensity;
            }
        }
    }

    return intensity;
}

/**
 * @return Returns the grid.
 */
public String[][] getGrid() {
    return grid;
} /**
 * @param grid The grid to set.
 */
public void setGrid(String[][] grid) {
    this.grid = grid;
} /**
 * @return Returns the points.
 */
public int[][] getPoints() {
    return points;
} /**
 * @param points The points to set.
 */
public int [][] setPoints(int[][] points) {
    this.points = points;
```

```
    return points;
} /**
 * @return Returns the xmax.
 */
public int getXmax() {
    return xmax;
}
/**
 * @return Returns the xmin.
 */
public int getXmin() {
    return xmin;
}

/**
 * @return Returns the ymax.
 */
public int getYmax() {
    return ymax;
}

/**
 * @return Returns the ymin.
 */
public int getYmin() {
    return ymin;
} /** }
```

Appendix E

Source Code for TreeConstruct.java

```
package AntSteiner.util;
import java.util.*;
import AntSteiner.*;

/**
 * Class: TreeConstruct.java
 * @author Anthony Blackman
 * @version April 19th, 2006
 */
public class TreeConstruct {
 /**This variable was used to store the reference to the random number
  * generator*/
 private Random rand;
 /**This variable was used to store length of the Steiner tree after prunning*/
 private int tLength = 0;
 /**This variable was used to store the number of vertices in the Steiner tree*/
```

```
private int tPoints = 0;
/**This data structure was used to store information regarding the type of vertex
 * an ant was currently at and whether the ant was confused at that vertex. */
private int [] info;
//**This data structure was used to store the relocation points of the
  *surviving ant.*/
//private int [] relocate; // This array stores the list of ants in the colony.
//private int [][] relocationPoints;
/**This data structure was used to store the mortality of the ants in the colony.*/
private int [] antID;
/**This data sturcture was used to store the vertices and edges of the Steiner tree.*/
private String [][] sTree;
/**This data structure was used to store the aerial view of the ant system.*/
private int [][] globalArray;
/**This tool is used to access the methods in the ant class.*/
private Ant tool;
/**This is an array of ants ie the ant colony.*/
private Ant [] colony;
/**This variable was used to store the identification of the ant who was met
 * if the current ant met an ant and it stored -1 if it did not met any ants*/
private int id;
/**This variable was used to indicate whether or not deconfusion had accorded*/
private int dCon;
/**This variable was used to store the identification of the ant current ant.*/
private int m;
/**This variable was used to store the number of ants still alive in the system.*/
private int ant_number = 0;
/**This data structure was used to store the current cooridinates each ant was at
```

```
 * during any instance of the program.*/
private int [][] antPoints;
/**
 * This is the default constructor.
 */
public TreeConstruct(){

    }
    /**
     *
     * @param terminals The terminals list.
     * @param grid The Hanan's Grid
     * @param intensity The intensity matrix
     * @return sTree The steiner tree.
     * */
public String [][] constructSteinerTreeByACO (int [][]terminals,
String [][] grid, double [][] intensity, int [] gridLimits){
    int n = terminals.length;

    tool = new Ant(grid.length);
    relocate = new int [3];
    antID = new int [n];
    rand = new Random();
    antPoints = new int [n][2];
    colony = new Ant[n];
    for(int i = 0; i < colony.length;i++){
        colony[i] =  new Ant(n, grid.length);
        // This makes a call to the
```

```
        //Ant Class overloaded constructor.;
        // This places a n by n String array in each ant and Visited
        // cites list. It also places the trail length of each ant also.
    }

    //This loops was used to store where the ants are at a given time.
    for(int i = 0; i < terminals.length; i++){
        for(int j = 0; j < 2; j++){
            antPoints[i][j] = terminals[i][j];
        }
    }

    // This loop was used to place an ant at there starting location.
    for(int i = 0; i < colony.length; i++){
    tool = tool.placeAnt(tool, colony[i],terminals[i][0], terminals[i][1], i);
    }

    ant_number = antID.length;
    for(int s = 0; s < antID.length; s++){
        antID[s]= 1; //This one indicates weather an ant is alive. When
                     //an ant at position i dies then Then the i will put a -1
                     //in position i to indicate the ant is dead. This
                     //This would prevent me from choosing dead ants to move.
    }


while (ant_number > 0)
```

```
{  info = new int [3];
   dCon = 0;
   m = rand.nextInt(antID.length); //generates a random number between 0
  //and number of ants.

  while (antID[m]== -1){// Keeps generate a random number until a live ant
                        //is found.
  m = rand.nextInt(antID.length); //generates a random number between
                                  //0 and number of ants.
 }
 //Move an ant that is still alive.
 info = tool.AntMove(tool, colony[m],antPoints[m][0], antPoints[m][1], m, grid,
 intensity, terminals, colony, antID, gridLimits);
 id = info[0]; // This id is used to indicate if an ant met another ant or not.
               // -1 means did not meet an ant.
 dCon = info[3]; //Indicates whether or not there was a deconfusion.
                 //1 means NO! and -1 means YES!

 while (id == -1){ //Move an ant that is still alive.
   info = tool.AntMove(tool, colony[m],info[1], info[2], m, grid, intensity,
                       terminals, colony, antID, gridLimits);
       id = info[0];
       dCon = info[3];
       System.out.println("\nWork now!!" + " id: " + id + " m: " + m);
       if (dCon == -1){
             break;
           }
       }
```

```
      int count = 0;
  while(dCon == -1 && id != -1 ){// if the ant was deconfused then move him.
    count ++;
    antPoints[id][0] = info[1]; //This now puts the ant on a new in his tabulist
    antPoints[id][1] = info[2]; //This now puts the ant on a new in his tabulist

      //Move an ant that is still alive.
      info = tool.AntMove(tool, colony[m],info[1], info[2], id, grid, intensity,
                          terminals, colony, antID, gridLimits);
      dCon = info[3];
      id = info[0];
      if (count > terminals.length){
          break;
       }
     System.out.println("dCon is : " + dCon);
      }


// This line says that if the ant meets another ant
// then he dies an copies his trail into the ant.
if (id != -1){// id not equal to minus one means that
              //the ant has met another ant.
colony[id]= tool.copyAntTour(colony[id], colony[m], grid.length);
  antID[m] = -1; // ant m is that ant that was killed
                //by ant id whom he met.
//tool.mergeTours(tool.getGlobalView(), colony[id].getTour(),id);
// This merges the tour of the ant with the global tour.
//tool.updateGlobalView(tool.getGlobalView(),m,id);
```

```
    System.out.println("\nAnt " + m + " met ant " + id);
    System.out.println("\nAnt " + m + " just died!\n");
   // relocate = tool.relocate (colony[id],  antPoints, grid);
   // antPoints[id][0] = relocate [1]; //This now puts the ant
                                       //on a new in his tabulist
   // antPoints[id][1] = relocate [2]; //This now puts the ant
                                       //on a new in his tabulist
  // id = relocate[0];
    ant_number--;
  //System.out.println(id + " " + ant_number);
  }
}


    globalArray = new int [grid.length][grid.length];
    globalArray = tool.getGlobalView();
    sTree = new String[grid.length][grid.length];


    for(int j = 0; j < sTree.length; j++){
         for (int i = 0; i < sTree.length; i++){
            sTree[j][i] = " ";
         }
     }

     for(int i = 0; i < grid.length; i++){
         System.out.print("\n");
         for (int j = 0; j < grid.length; j++){
```

```
            if (globalArray[i][j] != -1){
                System.out.print(globalArray[i][j]);
            }
            else{
                    System.out.print(" ");
                }
        }
    }
    System.out.println("\nThis is the Steiner Tree Before Pruning!\n\n");
    for(int i = 0; i < grid.length; i++){
        System.out.print("\n");
        for (int j = 0; j < grid.length; j++){
            if (globalArray[i][j] != -1){
                System.out.print(grid[i][j]);
                sTree[i][j]= "*";
            }
            else{
                System.out.print(" ");
                sTree[i][j]= " ";
            }
        }
    }
System.out.print("\n");
sTree = pRune(sTree, grid, gridLimits);
//This method was used to remove all
//leaves that were not terminals.
  tLength = 0;
  tPoints = 0;
```

```
for(int j = 0; j < sTree.length; j++){
   for (int i = 0; i < sTree.length; i++){
      if(sTree[j][i] == "*"){
          tLength++;
      }
      if(sTree[j][i] == "*" && (grid[j][i]== "o" || grid[j][i] == "x")){
          tPoints++;
      }
   }
}

tLength = tLength - tPoints; //This is the length of the tree formed.
//System.out.println("\nThe length after prunning is: "+tLength);

return sTree; }// End the tree construct.

/**
 *Description: This method was used to erase all non-terminal leaves from
 *the steiner tree that was created.
 *@param sTree The steiner tree with non terminal leaves
 *@param grid The Hanan's grid
 *@param gridLimits This array contains the values of the extremes of the
 *Hanan's grid
 *@return sTree The steiner tree with out non terminals leaves.
 */
public String [][] pRune (String [][] sTree, String [][] grid, int
[] gridLimits){
```

```
/* gridLimits[0] = xmax This is the value of the last row of the
Hanan's grid. gridLimits[1] = ymax This is the value of the last
column of the Hanan's grid. gridLimits[2] = xmin This is the value
of the first row of the Hanan's grid. gridLimits[3] = ymin This is
the value of the first column of the Hanan's grid.
*/
    int n = sTree.length * sTree.length;
for(int loops = 0; loops < n; loops++) {

    for(int row = gridLimits[2]; row <= gridLimits[0]; row++){
        for(int col = gridLimits[3]; col <= gridLimits[1]; col++){
            int degree = 0;
   if((grid[row][col] == "o" || grid[row][col] == "-"||
   grid[row][col] == "|" ) && sTree[row][col] == "*"){

           //This is for the top left had corner.
           if (row == gridLimits[2] && col == gridLimits[3]){
               if (sTree[gridLimits[2]][gridLimits[3]+ 1]!= " "){
                     degree++;
               }
               if (sTree[gridLimits[2]+1][gridLimits[3]]!= " "){
                     degree++;
               }
               }
            //This is for the botton right had corner.
            else if (row == gridLimits[0] && col == gridLimits[1] ){
                 if (sTree[gridLimits[0]][gridLimits[1] - 1]!= " "){
                       degree++;
```

```
        }
        if (sTree[gridLimits[0] - 1][gridLimits[1]]!= " "){
            degree++;
        }
        }

    //This is for the botton left hand corner.
    else if (row == gridLimits[0] && col == gridLimits[3] ){
        if (sTree[gridLimits[0] - 1][gridLimits[3]]!= " "){
              degree++;
        }
        if (sTree[gridLimits[0]][gridLimits[3]+ 1]!= " "){
              degree++;
        }
    }

   //This is for the top right hand corner.
   else if (row == gridLimits[2] && col == gridLimits[1] ){
        if (sTree[gridLimits[2] + 1][gridLimits[1]]!= " "){
              degree++;
         }
        if (sTree[gridLimits[2]][gridLimits[1] - 1]!= " "){
              degree++;
         }
        }

   //This is for the center of row 0.
     else if (row == gridLimits[2] && col <= gridLimits[1] ){
```

```
        if (sTree[gridLimits[2]][col - 1]!= " "){
              degree++;
        }
        if (sTree[gridLimits[2]][col + 1]!= " "){
            degree++;
        }
        if (sTree[gridLimits[2] + 1][col]!= " "){
            degree++;
        }
    }


//This is for the center of row sTree.length - 1.
  else if (row == gridLimits[0] && col <= gridLimits[1] ){
        if (sTree[gridLimits[0]][col - 1]!= " "){
             degree++;
        }
       if (sTree[gridLimits[0]][col + 1]!= " "){
             degree++;
        }
        if (sTree[gridLimits[0] - 1][col]!= " "){
             degree++;
        }
           }
//This is for the center of column 0.
  else if (row <= gridLimits[0] && col == gridLimits[3] ){
       if (sTree[row - 1][gridLimits[3]]!= " "){
             degree++;
```

```
            }
            if (sTree[row + 1][gridLimits[3]]!= " "){
                degree++;
            }
            if (sTree[row][gridLimits[3] + 1]!= " "){
                degree++;
            }
        }

      //This is for the center of column sTree.length - 1.
        else if (row >= gridLimits[2] && col == gridLimits[1] ){
            if (sTree[row - 1][col]!= " "){
                degree++;
            }
            if (sTree[row + 1][col]!= " "){
                degree++;
            }
            if (sTree[row][gridLimits[1] - 1]!= " "){
                degree++;
            }
            }

//This is for the center of the grid
else if (row >= gridLimits[2] && col <= gridLimits[1] && col >=
gridLimits[3]){
        if (sTree[row - 1][col]!= " "){
            degree++;
        }
```

```
        if (sTree[row + 1][col]!= " "){
             degree++;
        }
        if (sTree[row][col - 1]!= " "){
             degree++;
        }
        if (sTree[row][col + 1]!= " "){
             degree++;
        }
        }
        //If the degree of the node is 1 put a space.

//System.out.println("The degree is: " + degree + " row: " +row+ " col: "+col);

          if (degree == 1){
                //System.out.println("The degree is: " + degree);
                sTree[row][col] = " ";
            }
          }
       }
}

}

return sTree; }

    /**
     * @return Returns the tLength.
```

```
     */
    public int getTLength() {
        return tLength;
    }
    /**
     * @param length The tLength to set.
     */
    public void setTLength(int length) {
        tLength = length;
    }
    /**
     * @return Returns the id.
     */
    public int getId() {
        return id;
    }
    /**
     * @param id The id to set.
     */
    public void setId(int id) {
        this.id = id;
    }
}//End the class
```

Appendix F

Source Code for Ant.java

```
package AntSteiner;
import AntSteiner.util.*;
import java.lang.Math;
import java.util.*;

/**
 * Class: Ant.java
 * @author Anthony Blackman
 * @version April 10, 2006
 */
public class Ant {

/**xposition represents the x coordinate of the vertex*/

private int xposition;

/**xposition represents the y coordinate of the vertex*/
```

```
private int yposition;

/**This represents the length of an ants tour*/

private double tour_length = 0.0;

/**This stores the length of the tour/tree just created.*/

private double max = -1.0;

/**This stores the ants memory of is its trail.
*/ private String [][] tour;

/**This stores the list of terminals visited by the ant.*/

private int [] visited;

/**This stores the list of all the vertices in the Hanan's Grid*/
private int [][] starList;

/**This stores the list of all the Steiner points in the Hanan's
Grid*/
private int [][] steinerList;

/**This stores the probabilities the ant would go in the four
different directions*/

private double [] probability;
```

```
/**This is a reference to the Utilities class*/

private Utilities uTil;

/**This was used to mark steps of the ants*/

private int step = 0;

/**This return information related to the ants interaction at a
vertex.*/

private int [] alabel;

/**This return the cooridinates where the ant was deconfused and
 * whether it is still confused*/
private int [] confuse;

/**This return the cooridinates where the ant was relocated*/
private int [] relocate;

/**This is a reference to a random variable*/

private Random random;

/**This is the data structure used to store the arial view of the
ant system.*/
```

```
private int [][] globalView;

/**
 * This is the default constructor.
 **/
    public Ant(int m){
 uTil = new Utilities();

 globalView = new int [m][m]; //This array will contain a global view
                                        //of the system.
        for (int i = 0; i < globalView.length; i++){
            for(int j = 0; j < globalView.length; j++){
                globalView[i][j] = -1;
            }
        }
    }
/**
 * Discription: This is the over ridden constructor. The copies the Hanan Grid
 * into the Ant Tour so that the ant would walk on his copy of the grid. This
 * would allow me to get the path of any ant. That way I can print the path of
 * the late surviving ant's tour which would be the Steiner Tree for
 * that iteration.
 *
 * @param n This is the number of cities or number of ants
 * @param m This is the size of the grid.
 *
 */
    public Ant(int n, int m){
```

```
    tour_length = 0.0;
    tour = new String [m][m];

    visited = new int [n];
    for (int i = 0; i < tour.length; i++){
        for(int j = 0; j < tour.length; j++){
            tour[i][j]= " ";
        }
    }

    for(int j = 0; j < visited.length; j++){
        visited[j]= 0; //0 represents unvisited terminals and 1 represents
                       //visited terminals.
    }

    }
/**
 * Discription: This method was design to place an ant m to the starting
 * position of his Hanan's Grid. How ever this placement would be in his
 * tabu list. ie. This tour.
 * @param xposition This is the x cooridnate where the ant will be placed.
 * @param yposition This is the y cooridnate where the ant will be placed.
 * @param i this indicates which city got the ant.
 * @param tool This is a reference to the ant system.
 * @param m This is the ant number m that was place at the point (x,y) of
 * the Hanan's gird.
 * @return This returns the reference to an ant.
 */
```

```
    public Ant placeAnt (Ant tool, Ant m, int xposition, int yposition, int i){
        this.xposition = xposition;
        this.yposition = yposition;
        m.tour[xposition][yposition]= "a"; //this is the tour for ant m.
        tool.globalView[xposition][yposition]= i;
        m.setCity(i);

    return tool;
    }

/**
 * Discription: This method move an ant to an new position j based on the
 * desirability and trail intensity to move to j. This method moves one postition
 * at a time.
 * @param xposition This is the x cooridnate where the ant will be placed.
 * @param yposition This is the y cooridnate where the ant will be placed.
 * @param antIndex This indicates which ant was moved.
 * @param m This is the ant number m that was place at the point (x,y)
 * of the Hanan's gird.
 * @param tool This is a reference to the ant system.
 * @param grid This the Hanan's grid.
 * @param intensity This is the intensity matrix or intensity trail of the system.
 * @param terminals This is the terminal list of the system.
 *
 * @return alabel This is an array which indicates which vertex the
 * ant move to and what
 * type of vertex was it.
 */
```

```
public int [] AntMove(Ant tool, Ant m, int xposition,
int yposition, int antIndex, String[][] grid, double [][] intensity,
int [][] terminals, Ant [] colony, int [] antID, int[] gridLimits){
    int n = grid.length;
    int xPlus = xposition + 1;
    int yPlus = yposition + 1;
    int xMinus = xposition - 1;
    int yMinus = yposition - 1;


    random = new Random();
    probability = new double [4];
    alabel = new int [4];


    //This looks ahead in the same column for the next vertex.
    while(xPlus < grid.length && grid[xPlus][yposition] == "|"){
        xPlus++;
    }
    // This looks back in the same column for the next vertex.
    while(xMinus >= 0 && grid[xMinus][yposition] == "|"){
        xMinus--;
    }
    // This looks ahead in the same row for the next vertex.
    while(yPlus < grid.length && grid[xposition][yPlus] == "-"){
        yPlus++;
    }
    // This looks back in the same row for the next vertex.
```

```
    while(yMinus >= 0 && grid[xposition][yMinus] == "-"){
        yMinus--;
    }


    for (int k = 0; k < probability.length; k++){
            probability[k] = -1.0; // Some invalid probability number.
       }


//These if's assign the probablility of going in the four different paths
//if available.

 if(xPlus < grid.length){//This makes sure that the ant does not run
                        //off the grid.
if(m.tour[xPlus][yposition] != " "
|| tool.globalView[xPlus][yposition] == antIndex){
         probability[0] = 0.0;   // if visited probability equals 0.0.
 }
 else
 {
  probability[0] = uTil.selectionProb(xPlus, yposition, intensity,
  m, n, terminals, grid, colony, antID);
  }
 }

if(xMinus >= 0){//This makes sure that the ant does not run off the grid.
 if(m.tour[xMinus][yposition]!= " "
```

```
|| tool.globalView[xMinus][yposition] == antIndex){
          probability[1]= 0.0;
         }
         else
              {
probability[1] = uTil.selectionProb(xMinus, yposition, intensity, m,
n, terminals,grid, colony, antID);
              }

       }

if(yPlus < grid.length){//This makes sure that the ant does not run
off the grid. if(m.tour[xposition][yPlus]!= " " ||
tool.globalView[xposition][yPlus] == antIndex){
         probability[2]= 0.0;
     }
     else
     {
probability[2] = uTil.selectionProb(xposition, yPlus, intensity, m,
n, terminals, grid, colony, antID);
           }
       }

if(yMinus >= 0){//This makes sure that the ant
//does not run off the grid.

if(m.tour[xposition][yMinus]!= " " ||
tool.globalView[xposition][yMinus] == antIndex){
```

```
                probability[3]= 0.0;
                }

                else
                {
 probability[3] = uTil.selectionProb(xposition, yMinus, intensity, m, n, terminals,
                    grid, colony, antID);
        }

       }
        if (probability[0] < 0.0){
           probability[0] = 0.0;
        }

        if (probability[1] < 0.0){
               probability[1] = 0.0;
            }

        if (probability[2] < 0.0){
               probability[2] = 0.0;
            }

        if (probability[3] < 0.0){
               probability[3] = 0.0;
            }
double sum_probability = probability[0] + probability[1] +
probability[2]+ probability[3];
  //this is the total.
```

```
// This section takes care of the case when the ant gets confuse.
     if (sum_probability == 0.0)
     {
          alabel = deconfuse(m,terminals, grid, antIndex);
          alabel[3] = -1;
         //   terminals[antIndex][0] = alabel[1];
      //  terminals[antIndex][1] = alabel[2];
     System.out.println("\n\nAnt "+ antIndex + " had to be deconfused!");
     }
     else
     {
      probability[0] = probability[0]/ sum_probability;
      probability[1] = probability[1]/ sum_probability;
      probability[2] = probability[2]/ sum_probability;
      probability[3] = probability[3]/ sum_probability;

      //This loop finds the maximum probability.
      max = -1.0;
         int p = 0;
         while(p < 4){
          if (max < probability[p]){
              max = probability[p];
              p++;
          }
          else{
              p++;
          }
```

```
        }

 int choice = random.nextInt(4); //This generates a random number
                                 //between 0 and 4 not including 4.
        while(probability[choice]!= max){
            choice = random.nextInt(4);
        }
//System.out.print("\n\np0= " + probability[0]+ " p1= "
+ probability[1]+ " p2= "+ "probability[2]+ " p3= "+
probability[3]);

 //These if's move in the direction of the maximum probability
 //until u hit an x, o or run of the grid.
  if(choice == 0){
     step = xposition + 1;
      while (grid[step][yposition]== "|" && (step < grid.length)){
        m.tour[step][yposition]= "*";
        tool.globalView[step][yposition]= antIndex;
        step++;
      }
  if (grid[step][yposition]== "o" && tool.globalView[step][yposition] == -1){
     m.tour[step][yposition] = "*";
     tool.globalView[step][yposition] = antIndex;
     alabel[0]= -1;
     alabel[1] = step;
     alabel[2] = yposition;
   }// At this point we have to find out which ant's trail we just met.
else if(grid[step][yposition]== "o" &&
```

```
tool.globalView[step][yposition] != -1){

     m.tour[step][yposition] = "*";
     alabel[0]= tool.globalView[step][yposition];
     tool.globalView = mergeTours(tool.globalView, m.tour, antIndex);
     tool.globalView[step][yposition]= antIndex;
     alabel[1] = step;
     alabel[2] = yposition;
     tool.globalView = updateGlobalView(tool.globalView, antIndex, alabel[0]);

  }
   // At this point we have to find out which ant we just met.
     else if(grid[step][yposition]== "x"){
     m.tour[step][yposition] = "*";
     alabel[0]= tool.globalView[step][yposition];
   //tool.globalView = mergeTours(tool.globalView, m.tour, antIndex);
   //alabel[0] = getTheAnt(step, yposition, terminals);
     alabel[1] = step;
     alabel[2] = yposition;
     tool.globalView = updateGlobalView(tool.globalView, antIndex, alabel[0]);
    }
    //System.out.println("\n\nThe maximum [0] is: " + max);
     alabel[3]= 1; //Did not have a deconfuse.
 }
  else if(choice == 1){
     step = xposition - 1;
     while (grid[step][yposition]== "|" && (step >= 0)){
         m.tour[step][yposition]= "*";
```

```
        tool.globalView[step][yposition]= antIndex;
        step--;
            }
 if (grid[step][yposition]== "o" && tool.globalView[step][yposition] == -1){
      m.tour[step][yposition] = "*";
      tool.globalView[step][yposition]= antIndex;
      alabel[0]= -1;
      alabel[1] = step;
      alabel[2] = yposition;
 }// At this point we have to find out which ant's trail we just met.
else if(grid[step][yposition]== "o" &&
tool.globalView[step][yposition] != -1){

     m.tour[step][yposition] = "*";
     alabel[0]= tool.globalView[step][yposition];
     tool.globalView = mergeTours(tool.globalView, m.tour, antIndex);
     tool.globalView[step][yposition]= antIndex;
     alabel[1] = step;
     alabel[2] = yposition;
     tool.globalView = updateGlobalView(tool.globalView, antIndex, alabel[0]);

    }
 // At this point we have to find out which ant we just met.
    else if(grid[step][yposition]== "x" ){
      m.tour[step][yposition] = "*";
      alabel[0]= tool.globalView[step][yposition];
 //tool.globalView = mergeTours(tool.globalView, m.tour, antIndex);
 //alabel[0] = getTheAnt(step, yposition, terminals);
```

```
        alabel[1] = step;
        alabel[2] = yposition;
    tool.globalView = updateGlobalView(tool.globalView, antIndex, alabel[0]);
     }
    //System.out.println("\n\nThe maximum [1] is: " + max);
      alabel[3]= 1; //Did not have a deconfuse.
    }

     else if(choice == 2){
        step = yposition + 1;
    while (grid[xposition][step]== "-" && (step < grid.length)){
       m.tour[xposition][step]= "*";
       tool.globalView[xposition][step]= antIndex;
       step++;
    }
    if(grid[xposition][step]== "o" && tool.globalView[xposition][step]  == -1){
      m.tour[xposition][step] = "*";
      tool.globalView[xposition][step]= antIndex;
      alabel[0]= -1;
      alabel[1] = xposition;
      alabel[2] = step;

     }// At this point we have to find out which ant's trail we just met.
   else if(grid[xposition][step]== "o" &&
   tool.globalView[xposition][step] != -1){

      m.tour[xposition][step] = "*";
      alabel[0] = tool.globalView[xposition][step];
```

```
    tool.globalView = mergeTours(tool.globalView, m.tour, antIndex);
    tool.globalView[xposition][step]= antIndex;
    alabel[1] = xposition;
    alabel[2] = step;
    tool.globalView = updateGlobalView(tool.globalView, antIndex, alabel[0]);

}
 // At this point we have to find out which ant we just met.
  else if(grid[xposition][step]== "x" ){
       m.tour[xposition][step] = "*";
       alabel[0] = tool.globalView[xposition][step];
 //tool.globalView = mergeTours(tool.globalView, m.tour, antIndex);
 //alabel[0] = getTheAnt(xposition, step, terminals);
    alabel[1] = xposition;
    alabel[2] = step;
    tool.globalView = updateGlobalView(tool.globalView, antIndex, alabel[0]);
}
 //System.out.println("\n\nThe maximum [2] is: " + max);
 alabel[3]= 1; //Did not have a deconfuse.
}

else if(choice == 3){
    step = yposition - 1;
    while (grid[xposition][step]== "-" && (step >= 0)){
       m.tour[xposition][step] = "*";
       tool.globalView[xposition][step]= antIndex;
       step--;
    }
```

```
  if(grid[xposition][step]== "o" && tool.globalView[xposition][step] == -1){
  m.tour[xposition][step] = "*";
  tool.globalView[xposition][step]= antIndex;
  alabel[0]= -1;
  alabel[1] = xposition;
  alabel[2] = step;
  }  // At this point we have to find out which ant's trail we just met.
else if(grid[xposition][step]== "o" &&
tool.globalView[xposition][step] != -1){

  m.tour[xposition][step] = "*";
  alabel[0]= tool.globalView[xposition][step];
  tool.globalView = mergeTours(tool.globalView, m.tour, antIndex);
  tool.globalView[xposition][step]= antIndex;
  alabel[1] = xposition;
  alabel[2] = step;
  tool.globalView = updateGlobalView(tool.globalView, antIndex, alabel[0]);
 }
  // At this point we have to find out which ant we just met.
  else if(grid[xposition][step]== "x" ){
       m.tour[xposition][step] = "*";
       alabel[0]= tool.globalView[xposition][step];
      //tool.globalView = mergeTours(tool.globalView, m.tour, antIndex);
      //alabel[0] = getTheAnt(xposition, step, terminals);
       alabel[1] = xposition;
       alabel[2] = step;
 tool.globalView = updateGlobalView(tool.globalView, antIndex, alabel[0]);
 }
```

```
//System.out.println("\n\nThe maximum [3] is: " + max);
  alabel[3]= 1; //Did not have a deconfuse.
 }
 m.setCity(antIndex);
 } //This is for the else above.
return alabel;
}

 /**
  * Description: This methods was used to update the Global view of the system.
  * It replaced the tracks of the old or dead ant with the tracks for the new ant.
  * @param globalView This is the globalView array
  * @param antIndex This is the id of the dead ant
  * @param alabel This is the id of the alive ant
  * @return globalView This is the updated globalView array.
  */
 public int [][] updateGlobalView(int [][] globalView, int antIndex, int alabel){
     for(int i = 0; i < globalView.length; i++){
         for (int j = 0; j < globalView.length; j++){
             if (globalView[i][j] == antIndex){
                 globalView[i][j] =  alabel;
             }
         }
     }
    return globalView;
 }
/**
 * Description: This Method sets a city as visited for a give
```

```
 * ant tour.
 * @param city The city that was just visited.
 *
 */
public void setCity(int city){ visited[city]= 1; } /**
 * Description: This method look for the ant that is at
 * location (x,y) in the terminal array.
 * @param x This is the x cooridnate where the ant will be placed.
 * @param y This is the y cooridnate where the ant will be placed.
 * @param terminals This is the terminal list of the system.
 * @return index This indicates which ant was moved.
 * */
public int getTheAnt(int x, int y, int [][] terminals){
    int index = -20;

    for(int l = 0; l < terminals.length; l++){
        if (terminals[l][0] == x && terminals [l][1] == y){
            index = l;
            }
    }
return index; }

/**
 * Description: This method copies the tour of ant m
 * into the tour of ant m1.
 * @param m1 The ant who will do the copying
 * @param m The ant who got copied
 * @param n The number of rows of the ants tour
```

```
 * @return m1 The ant who did the copying
 * */
public Ant copyAntTour(Ant m1, Ant m, int n){

 for (int p = 0; p < n; p++){
    for(int q = 0; q < n; q++){
     if (m.tour[p][q] == "*" || m.tour[p][q] == "a" ){
      m1.tour[p][q]= "*";// This copies the ant m tour into ant id's tour.
            }
        }
    }
 // m1.tour_length = m1.tour_length + m.tour_length;
 // This adds the lenght of ant m's tour to ant m1's tour.

   return m1;
}

/**
 * Description: This method copies the tour of ant m into the
 * globalView (global Tour)
 * @param globalView The system view of the of the tour
 * @param tour The tour of the ant who connected with the system.
 * @return globalView The system view of the of the tour
 */

public int [][] mergeTours(int [][] globalView, String [][] tour,
int antIndex){
    for (int p = 0; p < globalView.length; p++){
```

```
        for(int q = 0; q < globalView.length; q++){
            if (tour[p][q] == "*" || tour[p][q] == "a" ){
               globalView[p][q]= antIndex;// This copies the tour
                                          //into the global view
                                          //of the tours.
            }
        }
    }
    return globalView;
}

/**
 * Descriptions: This method was used to deconfuse an ant
 * if he got trapped on his trail.
 * @param m This is the ant who was confused and who will be relocated.
 * @param grid This is the Hanan's grid
 * @param terminals This is the terminal set of the system.
 * @return confuse This is an array which contains the x and y
 * coordinates where the
 * ant was deconfused to
 * */
public int [] deconfuse (Ant m, int [][] terminals, String [][]
grid, int antIndex){
    int n = grid.length;
    int nSquared = n * n; // number of squares in the tour array.
    starList = new int [nSquared][2];
    confuse = new int [4];
    //This initizes the starList array.
```

```
    for(int t = 0; t < starList.length; t++){
        for(int s = 0; s < 2; s++){
            starList[t][s] = 0;
        }
    }

    int row = 0;
    for(int r = 0; r < grid.length; r++){
        for(int c = 0; c < grid.length; c++){
            if ((grid[r][c] == "o" || grid[r][c] == "x") && (m.tour[r][c] == "*"
            || m.tour[r][c] == "a")){
            starList[row][0]= r;// This loop tries to list all of the steiner points
            starList[row][1]= c; // in the tour of ant m.
            row++;
            }
        }
    }
    //System.out.println(row);
    //if (row > 0){
    int index = random.nextInt(row);
    confuse[1]= starList[index][0];
    confuse[2]= starList[index][1];
//  }
    //else{

    //confuse = closestPoint(confuse, starList, terminals, row);
    // This method finds the closest point. In the ants tabulist
    //to go to when it is confused.
```

```
//  }
    System.out.println("\nThe deconfuse point is "
    + "("+confuse[1] +","+ confuse[2]+")");
    confuse[0]= globalView[confuse[1]][confuse[2]];
    //confuse[0]= antIndex;
    confuse[3]= -1;
    return confuse;
}

/**
 * Descriptions: This method was used to relocate an
 * ant to a steiner point on his trail.
 * @param m This is the ant who was confused and who will be relocated.
 * @param grid This is the Hanan's grid
 * @param terminals This is the terminal set of the system.
 * @return relocate This is an array which contains the x and
 * y coordinates where the ant was deconfused too
 * */
public int [] relocate (Ant m, int [][] terminals, String [][]
grid){
    int n = grid.length;
    int nSquared = n * n; // number of squares in the tour array.
    steinerList = new int [nSquared][2];
    relocate = new int [4];
    //This initizes the starList array.
    for(int t = 0; t < steinerList.length; t++){
        for(int s = 0; s < 2; s++){
            steinerList[t][s] = 0;
```

```
        }
    }

int row = 0;
for(int r = 0; r < grid.length; r++){
   for(int c = 0; c < grid.length; c++){
     if (grid[r][c] == "o" && (m.tour[r][c] == "*" || m.tour[r][c] == "a")){
        steinerList[row][0]= r;// This loop tries to list all of the steiner points
        steinerList[row][1]= c; // in the tour of ant m.
     row++;
   }
 }
}

    relocate = closestPoint(relocate, steinerList, terminals, row);
    // This method finds the closest point In the ants tabulist
    // to go to when it is confused.

    System.out.println("\nThe relocation point is "
    + "("+relocate[1] +","+ relocate [2]+")");
    relocate[0]= globalView[relocate[1]][relocate[2]];
    return relocate;
}

/**
 * Description: This method attempts to find a steiner point that
 * is near to the other terminals. Then returns the steriner point
 * in the ant's tabulist nearest to the other ants.
```

```
 * @param confuse This is an array which contains the x and y coordinates
 * where the ant will be moved to
 * @param starList This is an array which store all of the points visited
 * by the other ants.
 * @param terminals This is the terminal set of the system.
 * @param row This is the row that indicates the depth of the starList set.
 * @return confuse This is an array which contains the x and y coordinates
 * where the ant
 * will be moved to
 */

public int [] closestPoint(int [] confuse, int [][] starList, int
[][] terminals, int row){
    int dist;
    int minDis = 1000000;

    for(int f = 0; f < row; f++){
        for (int g = 0; g < terminals.length; g++){
dist = Math.abs(terminals[g][0] - starList[f][0]) +
Math.abs(terminals[g][1] - starList[f][1]);
                if (dist <= minDis){
minDis = dist; //looks for the min distance between the steiner
                //points and the terminals.
confuse [1]= starList[f][0]; //This is the x cooridinate for the min
                            //distance.
confuse [2]= starList[f][1]; //This is the y cooridinate for the min
distance.
                }
```

```
        }
    }


    return confuse;
}

/**
 *
 * @return Returns the tour.
 */
public String[][] getTour() {
    return tour;
}

/**
* @param tour The tour to set.
*/
public void setTour(String[][] tour) {
    this.tour = tour;
}

/**
 * @return Returns the tour_length.
 */
public double getTour_length() {
    return tour_length;
}
```

```
/**
 * @param tour_length The tour_length to set.
 */
public void setTour_length(double tour_length) {
    this.tour_length = tour_length;
}

/**
 * @return Returns the visited.
 */
public int[] getVisited() {
    return visited;
}

/**
 * @param visited The visited to set.
 */
public void setVisited(int[] visited) {
    this.visited = visited;
} /**
 * @return Returns the xposition.
 */
public int getXposition() {
    return xposition;
} /**
 * @param xposition The xposition to set.
 */
```

```
public void setXposition(int xposition) {
    this.xposition = xposition;
} /**
 * @return Returns the yposition.
 */
public int getYposition() {
    return yposition;
} /**
 * @param yposition The yposition to set.
 */
public void setYposition(int yposition) {
    this.yposition = yposition;
} /**
 * @return Returns the max.
 */
public double getMax() {
    return max;
} /**
 * @param max The max to set.
 */
public void setMax(double max) {
    this.max = max;
} /**
 * @return Returns the globalView.
 */
public int [][] getGlobalView() {
    return globalView;
} /**
```

```
 * @param globalView The globalView to set.
 */
public void setGlobalView(int [][] globalView) {
    this.globalView = globalView;
}

}

//Experiment!!!!
/*if (sum_probability != 0.0){
     * double randomSubSum = sum_probability * random.nextDouble();
     //generates a random number between 0 and sum_probability.
        int j = 0;
        double p = 0.0;
        while(p <= randomSubSum){
            p = p + probability[j];
            System.out.println(" This is p: " + p + " This is randomSubSum "
            + randomSubSum + " when j = " + j);
        j++;
        }
     *
     */
//End Experiment!!!!
```

Appendix G

Source Code for Utilities.java

```
package AntSteiner.util;
import AntSteiner.*;

import AntSteiner.ui.*;

/**
 * Class: Utilities.java
 * @author Anthony Blackman
 * @version April 10, 2006
 */
public class Utilities {

/**This is the variable was used to store the desirability the ant
had to move to a given vertex.*/
    private double desirability;

/**This is the variable was used to store the shortest distance
  * between the ant's current location and the tabu list of the other ants.*/
```

```
private double psi;

/**This is the variable was used to store the intensity the of the trail.*/
private double intens;

/**This is the variable was used to store the change in intensity of the
 * edge of the grid.*/
 private double delta_tau;

/**This is the variable was used to store the rectilinear distance between
 * vertices on the grid.*/
private double rec_distance;

/**This data structure was used to store the current trail in an ant's tabu
 * list.*/
private String [][] tourTemp;

/**This is the variable was used to store the minimum distance between points in
 * the ants tabu list.*/
private double minimum = 1000000.0;

/**This is the variable was used to store the combination of desirability and
 * intensity*/
private double intense_desire;

/**This is the variable was used to store the power of the trail intensity.*/
private double alpha;
```

```
    /**This is variable was used to store the power of the desirability.*/
    private double bata;

    /**This is the variable was used to store the constant parameter in the
     * desirability formula.*/
    private double gamma;

    /**This is the variable was used to store the evaporation parameter for
     * the system.*/
    private double rho;

    /**This is the variable was used to store the quality of the trail.*/
    private int Q;


/**
 * This is the default constructor.
 */
public Utilities(){
    alpha = Double.parseDouble(InputUI.parameters[0]);
    bata= Double.parseDouble(InputUI.parameters[1]);
    rho = Double.parseDouble(InputUI.parameters[2]);
    gamma = Double.parseDouble(InputUI.parameters[4]);
    Q = Integer.parseInt(InputUI.parameters[5]);
}


/**
```

```
* Discription: This methods gets the intersity of the trail from x to y.
* @param x city
* @param y city
* @param intensity array
* @param m The ant m
* @return intense_desire selection probability for the specific edge.
*/
    public double selectionProb(int x, int y, double [][] intensity, Ant m,
    int n, int [][] terminals, String [][] grid, Ant [] colony, int [] antID) {

        if (m.getTour_length()== 0.0)
        {
            delta_tau = 0.0;
        }
        else
        {
            double c = m.getTour_length();
            delta_tau =  Q / c;
        }

        intens = ((double)1 - rho)*intensity[x][y] + rho * delta_tau;
        //This is the trail intensity.


        psi =  createPSI(colony, x, y, grid, antID);
        // This is the shortest distance from vertex i to all
        // the vertices in the tabulist of the other ants.
```

```
        desirability = 1 / (1 + gamma * psi);

        intense_desire = Math.pow(intens, alpha)*Math.pow(desirability, bata);


        return  intense_desire;
    }

    /**
     * This Method calculates the rectilinear distance between
     * city x and city y.
     * @param x city x
     * @param y city y
     * @param row
     * @param col
     * @return rec_distance
     */

    public double recDistance(int x, int y, int row, int col){
        rec_distance = Math.abs(x - row)+ Math.abs(y - col);
        return rec_distance;
    }


/**
 * Descriptions: This method was used to relocate an ant to a
 * steiner point on his trail.
 * @param colony The set of ants
```

```
 * @param grid The Hanan's grid
 * @param x cooridnate
 * @param y cooridnate
 * @return minimum This is the value of PSI
 * */
public double createPSI(Ant [] colony, int x, int y, String [][]
grid, int [] antID){
    int n = grid.length;
    double min;
    tourTemp = new String [n][n];

for(int antnum = 0; antnum < colony.length; antnum++){
    if (antID[antnum]!= -1){
        tourTemp = colony[antnum].getTour();
      for(int row = 0; row < grid.length; row++){
         for(int col = 0; col < grid.length; col++){
         // This line looks for all the x's and the o's in the
         //tabulist of the ant called antnum
         if((tourTemp[row][col] == "*" || tourTemp [row][col]== "a" )&&
           (grid[row][col] == "x" || grid [row][col]== "o")){
      //This finds the rectilinear distance between (x, y) and (row, col);
                min =  recDistance(x, y, row, col);
                     if(min < minimum){
                        minimum = min;
                    }}}}}}
    return minimum;
    }}
```

Appendix H

Source Code for Display.java

```
package AntSteiner.ui;
import java.awt.Color;
import hsa.Console;

/**
 * Class: Display.java
 * @author Anthony Blackman
 * @version April 10, 2006
 */
public class Display {
    /**This variable was used as a reference to the output console*/
    private Console c;
 /**
 * This is the default constructor.
 */
    public Display(){

    }
```

```
/**
 * Description: This method was used to display the Steiner Tree;
 * @param sTree The steiner tree
 * @param grid The Hanan's grid
 * @param tLength The length of the steiner tree.
 */

public void displayTree(String [][] sTree, String [][] grid,
int tLength, long took){
//int n = grid.length;

c = new Console(45, 100, "Ant Tour" + " "); // from the HSA Package.

c.print("\nThe Steiner Tree length is: " + tLength );
c.print("\n\nThe execution time was: " + took + " milliseconds. ");
  for(int i = 0; i < grid.length; i++){
    //  System.out.print("\n");
      c.print("\n");
       for(int j = 0; j < grid.length; j++){
       //  System.out.print(grid[i][j]+" ");
           if (sTree[i][j] == "*" || sTree[i][j] == "a"){
               c.setTextColor(Color.BLACK);
               c.print(grid[i][j]+ " ");
           }
           else
           {
               c.setTextColor(Color.WHITE);
```

```
                c.print(grid[i][j]+ " ");
            }
        }
    }
            //c.drawString ("k",30, 400);
}         //c.drawLine(10,10,10, 300); }

/**
 * Description: This method was used to display the Hanan's Grid;
 * @param grid The Hanan's grid.
 */
public void displayGrid(String [][] grid){
//   This prints the grid.
        c = new Console(45, 100, "Hanan Grid");
        for (int i = 0; i < grid.length; i++) {
        //System.out.print("\n");
            c.print("\n");
            for(int j = 0; j < grid.length; j++){
                //System.out.print(grid[i][j]+" ");
                if (grid[i][j]== "-" || grid[i][j]== "|"){
                c.setTextColor(Color.RED); //From the hsa Package .
                c.print(grid[i][j]+" ");
                }
                else {//System.out.print(grid[i][j]+" ");
                    c.setTextColor(Color.BLACK); //From the hsa Package .
                    c.print(grid[i][j]+" ");
                }
            } } }}
```

Appendix I

Source Code for the MST_via_MatrixTree_Nu.m

```
function tree_length = MST_via_MatrixTree_Nu (N)
   %Name: Anthony Blackman
   %Aim: The main purpose of this function is to calculate the length of the MST
   %     generated by a set of N randomly generated vertices. Using the
   %     Kirchoff's Matrix-Tree Theorem.
   %Date: January 12th 2006
   %Reason for Development: Algorithm for the author's Masters Thesis at UBC.

MinTreeValues = zeros(5,2)% This array will store the points to plot generated

VSet = zeros(N,3) % This array will store the Vertices generated

Tree_Vert = zeros(N,N)% This array will store the City Plan (ie. graph layout)

Distance_Matrix = zeros(N,N)% This array will store the rectilinear distance
                            % between Vertex i
```

```
                                % and Vertex j in position (i,j)

MTree = zeros(N,N)% This array will store the expressions generated by the
                  % Matrix-tree Theorem.
temp = zeros(N,1)  % Will be used the store the row sum.

    for k = 1:N
        x = randperm(N) % This function generates a random permutation of
                        % numbers 1 to N

        i = x(1)% This function select the kth number in the randomly generated
                % list integers
                % 1 to N. The i represents the ith row where the vertex will
                % be placed.

        y = randperm(N) % This function generates a random permutation of
                        % numbers 1 to N

        j = y(1)% This function select the kth number in the randomly
                % generated list integers
                % 1 to N. The j represents the jth column where the vertex
                % will be placed.

        Tree_Vert(i,j)=1 % Put a one in position (i,j) to indicate the presents
                         % of a vertex.

        VSet(k,1)=k % This line stores the vertex V_k in row k column 1
                    % of VSet
```

```
    VSet(k,2)=i % This line stores the row position of V_k in row k
                % column 2 of VSet
    VSet(k,3)=j % This line stores the column position of V_k in row k
                % column 3 of VSet
end % This ends the for loop.

for alpha = 1:N
    for Beta = 1:N
        Distance_Matrix(alpha, Beta) = abs(VSet(alpha,2)-VSet(Beta,2))
                                     + abs(VSet(alpha,3)-VSet(Beta,3))
        Distance_Matrix(alpha, Beta)= -Distance_Matrix(alpha, Beta)
        % The two statements above calculates the rectilinear distance
        % between vertex alpha and vertex bata then stores the value at
        % position (alpha, bata) of the Distance_Matrix. After this we
        % then multiply the value by -m where m is a parameter that
        % will be dealt with later.
    end % This ends the inner for loop
end % This ends the outer for loop.

for m = 1:5
 T= Distance_Matrix
    Distance_Matrix = m*Distance_Matrix

 MTree = exp(Distance_Matrix) %This raises each element of the
                              % Distance_Matrix to the power e
 MTree = -MTree %This puts the minus exp into the matrix

  for w = 1:N
```

```
        MTree(w,w) = 0 % Since distance between vertex one and vertex
                       % one is zero we would have a minus one
                       % at each element of the main diagonal since we took
                       % the exp of the matrix then time -1 and -e^0 is -1.
                       % However we dont need a one in that position.
     end

temp = sum(MTree) % This line stores the column sums of the matrix MTree in a
                  % vector called temp.
                  % In the thesis I said that the row sums should be zero
                  % but in this case the row sum = column sum because the
                  % the distance matrix is a symmetric matrix and A transpose = A.
temp = -temp

for p = 1:N
    MTree(p,p)= temp(p) % This line places the some of row p in position
                        % (p,p) of MTree. This makes the row sum of the
                        % matrix MTree to be all zero.
end % This line ends the for loop.

MTree(1,1) = MTree(1,1) + 1 % This line makes the sum of row one to be one.
                            % While the row sums of rows 2 to N are still zero.

Tree_Polynomial = det(MTree) % This line writes the determinant of
                             % MTree.

Tree_Polynomial= abs(Tree_Polynomial) % This function take the absolute value
                                      % of the determinant.
```

```
Log_Tree_Polynomial = log(Tree_Polynomial) % This line take the natural log
                                           % of the Tree polynomial

MST_function = Log_Tree_Polynomial / m % this lines is for the calculation
                                       % for ln(det(A(m)))/m.

MinTreeValues(m,1)= m
MinTreeValues(m,2)= abs(MST_function)
Distance_Matrix = T
end

maximum = max(MinTreeValues(m,2))

d=1:5 % This plots five points of the function for values of m = 1,2,3,4 and 5
plot(d,MinTreeValues(d,2)) xlabel('Value of M'), ylabel('MST Value')
title('Graph 1: Graph of MST via Matrix Tree Results for N')
Distance_Matrix = -Distance_Matrix % This gets back the actually distance matrix

MST = xmst(Distance_Matrix) % This passes the rectilinear distance matrix into the
                            % prim' algorithm

length = size(MST,2) % This calculates the edges in the array.
for e = 1 : length
     Tree(e)= Distance_Matrix(MST(1,e),MST(2,e)) % This stores the lengths of
                                                 % each edge in an array
end
```

```
MSTree_length = sum(Tree) % This finds the length of the array by summing all
                          % of the lengths of the edges.
% This is the end of to program.
```

Appendix J

Prim's Algorithm using Matlab

```
function E=xmst(D);
   %Name: Anthony Blackman
   %Aim: Minimum Spanning Tree using Prim's Algorithm.
   %Date: January 12th 2006
   %Reason for Development: Algorithm for the author's Masters Thesis at UBC.
   %Input: D Rectilinear distance matrix
   %        D(i,j) weight of edge i---j
   %        D must be symmetric
   %Output: E indices of edges of MST
   %Prim's Algorithm.

n=size(D,1); E=zeros(2,n-1); a=1:n-1; r=n; cnt=1; M=repmat(Inf,[2
n-1]); while length(a);
  [md,mi]=min([M(1,:);D(r,a)]);
  f=find(mi==2);
  M(1,f)=md(f);
  M(2,f)=r;
  [md,mi]=min(M(1,:));
```

```
  r=a(mi);
  E(:,cnt)=[r;M(2,mi)];
  a(mi)=[];
  M(:,mi)=[];
  cnt=cnt+1;
end;
```