Abstract

This project report is a description of the implementation of ACO Algorithms (Ant Colony Optimization) in java language and the use of them for solving the traveling salesman problem (TSP). It includes three deferent variations of AS algorithms: the simple AS, the Elitist AS, and the MinMax AS. At first there is an introduction to the TSP problem, and the AS algorithms and furthermore the detailed description of them and also a number of improvements to the program such as the local optimization and the use of treads. At the end there is the benchmarking of the three ACO algorithms with and without 2-opt, a discussion about our results. In the end we conclude and persectivate on our overall results and how we could add further improvements.

Preface

This project report focus on the theory behind the traveling salesman problem (TSP) wherein we try to find as optimal a solution as possible to the TSP, by using the ACO(Ant Colony Optimization) Min Max AS and EAS which is optimizations of the simple Ant Swarm algorithm (AS). In the start of the report there is given an introduction to the TSP, AS and Min Max AS and through this introduction a problem formulation will be specified. There will be elaborated about the basic theory behind the TSP and the theory behind how using an AS algorithm can be a way to solve the TSP problem. Hereafter it will be explained how to implement an AS algorithm and how we have done it in our program.

After describing how to implement a simple AS algorithm there is explained different variations of the AS that can be made to improve the results of the algorithm. The last of these variations that are explained is the Min Max AS. Hereafter we explain the theory and implementation of other improvements made to our code such as local optimization and threads. Next our test results from the program is benchmarked on different TSP instances and compared to other results found by similar ACS algorithms on the same instances. We will end with

discussing these results which ends up with a conclusion about the functionality of our program and how further improvements could be implemented.

This project has been made by Asger Lisborg and Aristeidis Stroumpos with Keld Helsaugn as supervisor and is to be considered as the bachelor report of Asger Lisborg. To read and understand this report it is suggested to have at least a basic understanding about programming algorithms and a basic understanding of the java programming language. Also some understanding of math functions should be needed.

The code product that we have made in our project has been strongly inspired by the code described in "*Ant Colony Optimization*"[DT04] at chapter 3 made by Marco Dorigo and Thomas Stützle. We have used this text primarily to construct our AS, the EAS variation and Min Max AS variation. Besides our own code we have made used stdDraw and DrawTour created by Keld Helsgaun to visualize our code.

Table of Contents:

1: Introduction

1.1: Problem specification:

In this introduction we narrow us down towards the focus of our project and the requirement specifications of our program.

The hive mind behavior of ants is one of the most complex social structures amongst insects. Within an ant colony the ants perform a complex set of behaviors which can be considered almost algorithmic. One of the most complex of these behaviors is their highly advanced foraging behaviors, where the ants manage to communicate between each other very optimal tours towards potential food sources, by the use of pheromones (BTD, 1999: 25-26).

By the use of computers now a day, it has become possible to remake this behavior as an algorithm, performing an artificial version of the behavior almost identically to how the ants forage in real life. This algorithm can be used in problem solving within many different algorithmic problems such as Sensor Communications problems, Communication network problems and the travelling salesman problem(DB, 2006:). Amongst these problems we have chosen to focus on solving the travelling salesman problem(TSP) to the best of our ability using an artificial ant foraging algorithm. Such an algorithm within computer science, is known as ACO.

We are with our project not trying to solve any society-specific problems, since that would require making a TSP algorithm, better than the current state-of-the-art TSP solving algorithms. Since there have been made many other projects about ACO before it can't be said we are trying to find any kind of new knowledge. What we have tried to do is making a java ACO application an explain how to implement it. So our targeting group is everyone who is interested in ACO as a TSP solution algorithm and wish to know how to implement it with java code.

1.2: TSP and ACS Algorithms introduction:

TSP is the problem of how to find the shortest route possible in the shortest amount of time, amongst several points on a map and return to the starting city (GHL, 1994: 2). It is an interesting problem to solve with a ACO algorithm as a heuristic because larger instances can't be solved with the use of non-heuristic algorithms. There exist many variations of the ACO algorithm, where the standard algorithm is called AS(Ant Swarm). Basically this algorithm includes a number of ant objects who deposit an amount of pheromone everywhere they go, which they use to

communicate with each other to find better tours. The ants do not always choose the tour with the most pheromone and therefore sometimes explore new routes. We have chosen in our project to implement AS and two different variations of it called EAS and MMAS. EAS stands here for Elite Ant Strategy and MMAS for Minimum Maximum Ant System. In EAS the ant that found the best tour at each iteration, is named the elite ant and deposits extra pheromone on the edges of its tour to increase the possibility of the other ants choosing them.

In MMAS only one ant deposit pheromone, a minimum and maximum value to the pheromone is introduced and occasionally the pheromones are reinitialized. Currently MMAS is considered one of the best variations of the AS, since it for most instances provides the tour closest to the optimum of the current ACO variations. We therefore consider MMAS our final programming product in this report. It can be noted that many consider the best extension to be the ACO extension named ACS, because it finds almost as good a tour but does so in less time.

Besides these extensions of the AS we will try to further improve our results and the speed of our program by other improvements. Most noticeably we have implemented 2-opt, 2.5-opt and threads to improve solution quality and speed. We implement 2-opt and 2.5-opt to further improve the quality of our tours at each iteration at the cost of the speed of our program. We are implementing threads because we want ants to construct and improve their tours in parallel to create a speedup, depending on how many cores the computers have that runs the program.

1.3: Requirement Specifications

We want here to describe what our program should be able to do to achieve our goal, which is to make as optimal an ACO algorithm, as possible. First we will start with listing the two main parameters that the quality of a TSP solving algorithm is measured:

- Have as good a solution as possible for the instance.
- Be able to solve a TSP problem as fast as possible.

To achieve these two goals we wish to implement:

- Have implemented a simple AS program to develop further with variations.
- Implement the extensions of the EAS and MMAS to improve our results.

- Perform local optimization as fast as possible using 2-opt and 2.5-opt on the tours we found to improve them even further.
- Improving our program speed by only check the nearest neighbors for feasible cities to go to when our ants are constructing the tour.
- Have precalculated matrixes to improve running time.
- Have tour construction and local optimizations running as threads.

By implementing all these improvements we should be able to find tours close to the optimal, on a reasonable sized instance. Besides improving the quality of the tour and the run time, some other criterions for a successful implementation of our program is:

- Print out a graphic Visualization of our tour.
- Implement both AS, EAS and MMAS in the same program so we can use three different extensions of the AS algorithm.
- Be able to parse parameters in the program for several variables such as ants and filename.
- Have as small and understandable code as possible needed to implement our product.
- Be able to read an instance from a tsp file getting the x and y coordinates from it.
- Be able to compute an initial tour with the Nearest Neighbor Algorithm for initialization.

2: Theory

2.1: TSP

Before beginning to explain the ant colony system algorithm and the different extensions of it, we need to first explain the travelling salesman problem.

The TSP is the problem of how to compute the shortest route between n number of cities where you have to visit them all and return to the city you started travelling from. It can be noted that the distances between the cities are in Euclidean distances. This can be illustrated with a travelling salesman, who has to visit a number of cities to sell his wares. When he has visited a city and sold his wares he don’t need to visit it again and when all cities are visited he wish to return back to his

home city (GHL, 1994: 2). It sounds simple but the problem is that it is hard to solve for a large instance, such as a TSP instance of 20 or more cities, in a satisfactory amount of time.

2.1.1: Enumeration Algorithm

The intuitively most logical and simple algorithm to use for solving the TSP is named the enumeration algorithm. What this algorithm does is that it checks out all possible combinations of tours when you have n cities and returns the best tour that it found. By using this relatively simple algorithm the optimal tour is always found for the instance. The problem with this algorithm is that the running time grows aggressively depending on the number of cities and quickly becomes more than any computer can handle. What causes this is that for a large instance there are an astronomical number of combinations of tours which the computer has to compute. Illustrated underneath is a table showing the growth (K, 2000: chapter 2.1):

$N = 3$	$1\ routes$
$N = 4$	$3\ routes$
$N = 5$	$12\ routes$
$N = 6$	$60\ routes$
$N = 7$	$360\ routes$
$N = 8$	$2.520\ routes$
$N = 9$	$20.160\ routes$
$N = 10$	$181.440\ routes$
$N = 11$	$1.814.400\ routes$
$N = 12$	$19.958.400\ routes$
$N = 13$	$239.500.000\ routes$
$N = 14$	$3.113.500.000\ routes$
$N = 15$	$43.589.000.000\ routes$
$N = 17$	$10.461.000.000.000\ routes$
$N = 19$	$3.201.200.000.000.000\ routes$
$N = 21$	$1.216.500.000.000.000.000\ routes$
$N = 23$	$562.000.000.000.000.000.000\ routes$
$N = 25$	$310.220.000.000.000.000.000.000\ routes$
$N = 30$	$4.420.900.000.000.000.000.000.000.000\ routes$
$N = 35$	$147.620.000.000.000.000.000.000.000.000.000.000\ routes$
$N = 40$	$10.019.900.000.000.000.000.000.000.000.000.000.000.000.000\ routes$

Figure 2.1 from slideshow by Keld Helsgaun(K, slide 12)

The reason for this exponential growth in number of possible tours is that the number of possible routes for an instance follow the formula $\frac{(n-1)!}{2}$. Therefore the amount of checks the program has to do is found by using combinatorial mathematics, to find and compare all possible tour with n cities. The reason why it is not n! but (n-1)! is that the starting city can always be predefined and

the reason why the result is divided by 2 is because tours are symmetric (P, 1994: p 1). Therefore an instance of just 21 cities would take several years for a enumeration algorithm program to find the optimal tour.

2.1.2: Heuristic algorithms

It is possible to get good results for large instances by using heuristic algorithms, though not always the optimal tour. A heuristic TSP algorithm tries to construct a tour, which is as close to the optimal as possible within the algorithms limits. The reason for using heuristics is to speed up the tour construction to an acceptable time frame (OP, 1996: p. 3). So the challenge in the TSP is to make a heuristic algorithm that finds a tour as close to the optimal as possible and does so in a small amount of time.

There are many different Heuristic algorithms such as Nearest Neighbor, Greedy algorithm, Insertion Heuristics, 2-opt and more(N, 2003: p. 2). All of these different algorithms falls into two different categories of heuristic tour construction algorithms. Approximate algorithms, where you create a tour gradually such as the Nearest Neighbor algorithm and Tour Improvement algorithms that improve on a predefined tour, for an example with 2-opt. In our program we use both since we first finds a tour using a ACO algorithm and then tries to improve it with 2-opt which makes our algorithm a combinatorial tour construction algorithm, meaning it uses both an approximate algorithm and tour improvement algorithm. (FM, 1996: p. 1).

2.1.3: Nearest Neighbor Algorithm:

One of the simplest of the heuristic algorithms are known as the Nearest Neighbor, where the basic move is that each time visit the closest city that has not been visited and after you visit all return to the starting city. The running time of the nearest neighbor algorithm will not rise as aggressively when the size of the instance increase, than with the enumeration algorithm. The way the Nearest Neighbor Algorithm functions is:

1. Define starting city.

2. Find the nearest unvisited city and go there.
3. Are there any unvisitied cities left? If yes, repeat step 2.
4. Return to the first city.

We have through testing found that the Nearest Neighbor algorithm is usually about 30% away from the optimal route, depending on the instance. The gap from the optimum varied on the instances we tried it on from 20% to 100% from the optimum. In conclusion it is not a very effective algorithm for finding an optimal tour, but it is useful to create an initial tour that can be used in our ACO to compute the initial pheromone levels in our AS and its extensions. We will explain how in the AS theory (N, 2003: p. 1).

In comparison with the enumeration and Nearest Neighbor Algorithm, the MMAS algorithm should be able to find a tour with under 2% difference from the optimal on most instance smaller than about 1000 (DS, 2004: 92).

2.2: Ant Theory

2.2.1: Ant Swarm algorithm tour Construction

We will here give a description of the theory for the ant swarm intelligence program. The basic principle is that you have a number of ants who are trying to find the best possible route by communicating with each other through pheromones. It can be noted that the AS algorithm is basically a greedy tour construction algorithm except for the fact that it also uses pheromones to communicate a better tour (DS, 2004: p. 71).

An ant swarm intelligence program runs a number of iterations and at each iteration it constructs a new tour for each ant. What happens at every iteration is that a method controlling the decision of where an ant should go next is called a number of times equal to the number of cities-1, multiplied with the number of ants. For each call of this method the current ant adds the next city to their tour and registers it as visited.

2.2.2: Pheromone update theory

After all ants have constructed a complete tour and returned to their home city, a pheromone trail is deposited at each edge between every two cities where each ant have been and some of the old pheromone evaporates (DS, 2004: 105). The amount of pheromone that the ants deposit depends on the length of the tour they found, so the shorter the tour length the more pheromone they will deposit. That means that the ants will be more inclined to follow the edges of a good tour, because there was deposited more pheromone on, next time the iteration runs. Also the bad tours that the ants found slowly evaporate so in the end the ants mainly favor the better edges when constructing a tour.

2.2.3: Ant Swarm algorithm Behavior

As the iteration runs for several times a change from an explorative phase can be observed, where the ants are more inclined to try new routes, because the pheromones values are at this time about the same amount between the edges. This means that the ants sometimes chose to go to a city with less pheromone than the one with the most, thus trying new combinations of routes. Which city the ant should go to next is decided by computing a matrix, with both columns and rows equal to the total number of cities, called choice_info that contains information about how big a chance there is for an ant in a current city i to go to any other of the cities.

2.2.4: Calculating Choice Info matrix:

To compute the choice info you need the pheromones and a heuristic factor that has been precalculated. The formula for calculating the heuristic factor for each city is:

Function 2.1: $n_{ij} = 1/d_{ij}$

Where n_{ij} is the heuristicfactor for the edge between cities i and j and d_{ij} is the distance between the two cities. So the smaller the distance between the two cities are, the greater chance there is for the ant to go there.

To be able to compute the initial values of choice info, before the tour construction starts, we need an initial value for the pheromones which we set to:

Function 2.2: $t_{ij} = t_0 = m/C^{nn}$

So every value in our pheromone matrix t_{ij} is set to the initial value t_0 which is:

Function 2.3: m/C^{nn}

Here m is the number of ants used in the algorithm and C^{nn} is the length of the tour found by the nearest neighbor algorithm. It is possible to use another approximate algorithm than the nearest neighbor algorithm to initiate the pheromones (DS, 2004: 70).

Now that we know how to initiate the pheromone trail and the heuristicfactor, we can look at how the choice information is calculated for the ants. The function for calculating the choice info is:

Function 2.4: $ci_{ij} = [t_{ij}]^{\alpha} + [n_{ij}]^{\beta}$

Where ci_{ij} is the choice information used for deciding which city j an ant at city i should go to, α is a parameter which decides how much weight the choice info should put on pheromone and β is the parameter controlling how much weight the ant should put on the heuristicfactor.

2.2.5: Ant Swarm algorithm decisionRule:

Now that the choice info is computed it can be used to decide where the ant should go next. when in a city I the probability p_{ij}^{k} for going to a city J, can be calculated by using the formula:

Function 2.5: $p_{ij}^{k} = \frac{[t_{ij}]^{\alpha}+[n_{ij}]^{\beta}}{\sum l \in N_i^k [t_{ij}]^{\alpha}+[n_{ij}]^{\beta}}, if\ J \in N_i^k$

Where N_i^k is the feasible neighborhood which an ant k at city I can go to, meaning every city that hasn't been visited by the ant already. So the function divides the choice info of a city i to a city j with the sum of the probability of all cities in the N_i^k (DS, 2004: 70).

The way to decide which city an ant should go to next when on a city i is to generate a random number between 0 and the value of sum probability of all cities in a feasible neighborhood. You then take the choice info for every edge of cities within the feasible neighborhood and add it to a variable, until the variable becomes larger than the random number. When this happens you go from city i to the city j, which had the choice info that made the variables value become larger than the random generated number. By doing this the ants will always have a higher probability to go to a city j which has a high choice info but sometimes it will go to a city j with a low choice info, thus exploring new possible routes (BTD, 1999: pp. 42-43).

2.2.6: AS Pheromone Update Algorithm:

After a tour has been constructed for each ant the pheromones are updated according to the deposit and evaporation rules. First pheromone evaporation is called which sets the pheromone on every edge equal to:

Function 2.6: $t_{ij} \leftarrow (1-\rho) * t_{ij}$

Where p is a parameter in the program that decides how big a percentage of the pheromone that should be evaporated. We have set it to 0.5 in our simple AS as suggested by Dorigo and Stützle (DS, 2004: p. 71). Hereafter the deposit pheromones are called which uses the function below to deposit pheromone:

Function 2.7: $t_{ij} = t_{ij} + \sum_{k=0}^{m} \Delta t_{ij}^{k}$, $\forall (i,j) \in L$ (DS, 2004: 72).

Where Δt_{ij}^{k} is the amount of pheromone that the ant should put on the edge between i and j. This is done for k=0 to m where m is the number of ants. This pheromone deposit is done on every edge each ant has crossed in its tour. Therefore the rule for depositing pheromone is:

2.8 Function: $t_{ij}^{k} = \begin{cases} 1/C^{k}, & if\ arc\ (i,j)\ belongs\ to\ T^{k} \\ 0, & Otherwise; \end{cases}$

Where T^{k} is the tour of the ant and C^{k} is the length of the tour.

3: Ant Swarm implementation

We will now explain how we have implemented a simple AS program solution.

The two classes we have are ASTSP and ANT. First the main method initialize the parameters for the program (such as m and the instance which include n and the x, y coordinates) and creates a new object of AsTSP. The constructor of the AsTSP class calls the Method initializeData, runs the tour construction loop and updates program statistics, pheromones and choiceInfo. Underneath is a skeleton of the ASTSP constructor in our program:

AsTSP Constructor:

InitializeData();

While (Terminate=true) do {

ConstructSolution();

UpdateStatistics();

ASPheromoneUpdate();

ComputeChoiceInfo();

}

3.1: InitializeData

Initialize Data makes the initial calculations for central variables.

InitializeData methods:

ReadInstance();

ComputeDistances();

ComputeNearestNeighborList();

ComputeNearestNeighborTourLength();

InitializePheromones();

InitializeHeuristicFactor();

ComputeChoiceInformation();

InitializeAnts();

The first data that are initialized is n and the x, y coordinates for the points by reading the tsp file in the readInstance method.

```
1.  public void ReadInstance(String fileName)
2.     {
3.        File file = new File(fileName);
4.        try {
5.           Scanner instanceScanner = new Scanner(file);
6.           n = instanceScanner.nextInt();
7.           double[] x = new double[n];
8.           double[] y = new double[n];
9.           this.x = x;
10.          this.y = y;
11.          for (int i = 0; i < n; i++) {
12.             int j = instanceScanner.nextInt() - 1;
13.             x[j] = instanceScanner.nextDouble();   //reading coordinates
14.             y[j] = instanceScanner.nextDouble();
15.          }
16.       } catch (FileNotFoundException ex) {
17.       getLogger(AsTSP.class.getName()).log(Level.SEVERE, null, ex);  }
18.    }
```

To do this we first create a Scanner object so we can read from the file in line 5. Then we get the number of cities (n) in line 6 by scanning the first integer in the file. Hereafter at line 11-14 we read every x and y coordinate for every city within the instance, by running the scan until a variable becomes larger than the n number of cities. We save the coordinates in two arrays named x[] and y[].

Next initializeData calls the ComputeDistances which computes a matrix with the distance from all cities to all other cities.

```
1.    public void ComputeDistances(double[] x, double[] y)
2.    {
3.       for ( int i=0; i<n; i++)
4.          for ( int j=i+1; j<n; j++) {
5.             double dx = x[i] - x[j], dy = y[i] - y[j];
```

```
6.              distance[i][j] = (int) (Math.sqrt(dx * dx + dy * dy)+0.5);
7.              distance[j][i]=distance[i][j];
8.          }
9.      }
```

As can be seen in line 2-7 of the computeDistance method, all distances for all city to all other cities are calculated and put into a distance matrix.

Next the NearestNeighborList matrix is filled, which includes for each city the nn closest cities sorted by distance. So the nearest neighbor is always the next in the row and the next nearest neighbor is the next after and so on.

```
1.  public void ComputeNearestNeighborLists()
2.      {
3.          int[][] sorted_dis = new int[n][nn];
4.          boolean[] sorted = new boolean[n];
5.          boolean  swapped;
6.          for (int i=0; i<n; i++)
7.              for(int j=0; j<nn; j++) {
8.                  if (i!=j) {
9.                      nnList[i][j]=j;
10.                     sorted_dis[i][j]=distance[i][j];
11.                     sorted[j]=true;
12.                 }
13.                 else if(i==j && (j+nn)<n){
14.                     nnList[i][j]=j+nn;
15.                     sorted_dis[i][j]=distance[i][j+nn];
16.                     sorted[j+nn]=true;
17.                 }
18.                 else if(i==j && (j+nn)>n){
19.                     nnList[i][j]=nn-j;
20.                     sorted_dis[i][j]=distance[i][nn-j];
21.                     sorted[j-nn]=true;
22.                 }
23.             }
24.         for (int i=0; i<n; i++) {
25.             do {
26.                 swapped = false;
27.                 for(int j=1; j<nn; j++) {
28.                     if (sorted_dis[i][j-1]>sorted_dis[i][j]) {
29.                         int temp = sorted_dis[i][j];
30.                         int temp2 = nnList[i][j];
31.                         sorted_dis[i][j] = sorted_dis[i][j-1];
32.                         nnList[i][j] = nnList[i][j-1];
```

```
33.                 sorted_dis[i][j-1] = temp;
34.                 nnList[i][j-1] = temp2;
35.                 swapped = true;
36.             }
37.           }
38.       } while (swapped);
39.    }
40.    for (int i=0; i<n; i++) {
41.       for (int j=0; j<n; j++) {
42.          sorted[j]=false;
43.       }
44.       for (int j=0; j<nn; j++) {
45.          sorted[nnList[i][j]]=true;
46.       }
47.       sorted[i]=true;
48.       for(int j=0; j<n; j++) {
49.          if (distance[i][j] < sorted_dis[i][nn-1] && sorted[j]==false) {
50.             sorted[nnList[i][nn-1]]=false;
51.             sorted_dis[i][nn-1]=distance[i][j];
52.             nnList[i][nn-1]=j;
53.             sorted[j]=true;
54.             for (int s=0; s<nn-1; s++) {    //sort function
55.                if(sorted_dis[i][s]>sorted_dis[i][nn-1]) {
56.                   int temp=sorted_dis[i][nn-1];
57.                   int temp2=nnList[i][nn-1];
58.                   for (int d=nn-1; d>s; d--) {
59.                      sorted_dis[i][d]=sorted_dis[i][d-1];
60.                      nnList[i][d]=nnList[i][d-1];
61.                   }
62.                   sorted_dis[i][s]=temp;
63.                   nnList[i][s]=temp2;
64.                   break;
65.                }
66.             }
67.          }
68.       }
69.    }
```

We need 3 matrixes to calculate our NearestNeighborList. First we need a matrix called sorted_dis that will include the nn closest distances, for each city n. Then we need an index that shows which city j represents each distance in the sorted_dis matrix called nnList. Last we need a boolean array called sorted which for each city shows if it's distance is within the sorted_dis. From line 1 to line 23 these matrixes are created and initialized. At line 24-39 we sort the sorted_dis array. Every change to the sorted_dis array is also applied to the nnList index. After this all the cities are

checked if they have a smaller distance than the last city of the nnList and if they do they take a position in the sorted_dis and the largest distance is deleted. So at the end we have a nnList[n][nn] filed with the nearest neighbors for every city i.

We now calculate the nearest neighbor tourLength (nntl). It is implemented as described in subchapter 2.1.3.

Hereafter we calculate the initial pheromone trail.

```
1.  public void InitializePheromoneTrail() {
2.          for (int i=0; i<n; i++)
3.             for(int j=i+1; j<n; j++) {
4.                pheromone[i][j] = (double)m / (double)nntl;
5.                pheromone[j][i] = pheromone[i][j];
6.             }
7.  }
```

As can be seen in line 3-6 we initialize every starting pheromone value for every edge to the same value, depending on the number of ants and length of the Nearest Neighbor Tour length. We now compute a heuristic factor matrix with all the heuristic information for every edge which we will use to compute the choice info.

```
1.  public void InitializeHeuristicFactor()
2.     {
3.        for (int i=0; i<n; i++)
4.           for(int j=i+1; j<n; j++) {
5.              heuristicFactor[i][j] =  1.0 / (distance[i][j]+0.0000001);
6.              heuristicFactor[j][i]=heuristicFactor[i][j];
7.           }
8.     }
```

As can be seen from line 3-7 we calculate the heuristic information for every edge depending on the length of the edge as described in formula 2.1. It can be marked that at the formula we add 0.0000001 at the distance to prevent division with 0.

Hereafter we call ComputeChoiceInfo which combines the heuristic factor with the pheromone trail to compute the ChoiceInfo matrix, which describes the preference of an ant in a city i to go to a city j.

```
2.  public void ComputeChoiceInformation()
```

```
3.    {
4.       for (int i=0; i<n; i++)
5.          for(int j=i+1; j<n; j++) {
6.             choice_info[i][j] = Math.pow(pheromone[i][j], alpha) *
    Math.pow(heuristicFactor[i][j], beta);
7.             choice_info[j][i] = choice_info[i][j];
8.          }
9.    }
```

So as can be seen in lines 4-7 the initial choice info for every edge is calculated as described in function 2.3. The choiceInfo is recomputed after every pheromone update at every iteration. We now got all the information needed to initialize the ants.

```
1.  public void InitializeAnts()
2.    {
3.       for (int i=0; i<m; i++)
4.          ants[i] = new Ant(n, distance, choice_info, nnList, nn, local);
5.    }
```

What we do in line 3-4 is that we create a number of new ant objects equal to the value of m. Each of these ant objects are given the n, m, distance, choice info and the nearest neighbor List. This info is used within each ant to contruct a tour.

At last we Initialize the statistic needed in the program such as the best so far tour.

After all these Initializations the ASTSP constructor starts the construct solution method of the class ANT so each ant will construct a tour. This method will be called for each ant as many times as the defined number of iterations.

3.2: AS construct solution implementation:

When the Construct Solutions method of the ANT class is called a random city is set to be the starting city for the tour and is marked as visited. Then a loop starts which each time add one city to the tour array. Every time it calls the Decision rule method to decide which city the ant should go next. At the end the ant returns to the starting city and local search is called to improve the tour if activated.

```
1.  public void ConstructSolutions()
2.  {
3.     for (int i=0; i<n; i++)
4.        visited[i]=false;
5.
6.     int r = (int) (rand.nextDouble() * n);
7.     visited[r]=true;
8.     tour[0]=r;
9.     index[r]=0;
10.
11.    for (int i=1; i<n; i++) {
12.       //ants[k].ASDecisionRule(i);
13.       NeighborListASDecisionRule(i);
14.    }
15.    tour[n]=tour[0];
16.    if (local==1) {
17.       Opt2();
18.       Opt2_5();
19.
20.    }
21. }
```

The constructSolution is implemented in a way that all ants construct their tours sequentially, it should be noted that if the construct solutions is moved to the ASTSP class it is easy to make the ants construct their tours in parallel.

3.2.1: NeighborListASDecisionRule Implementation:

The NeighborListASDescisionRule decides which city the ant should go to next, using the roulette wheel selection. So the next city is chosen by random chance, influenced by the amount of choice info on the edges of cities within a feasible neighborhood (DS, 2004: p. 107).

```
1.  public void NeighborListASDecisionRule(int step)
2.  {
3.     int c, j;
4.
5.     double sum_probabilities;
6.     double[] selection_probability= new double[n];
7.     double r, p;
8.
9.     c = tour[step-1];
```

```
10.       sum_probabilities=0.0;
11.
12.       for (j=0; j<nn; j++) {
13.          if ( visited[nnList[c][j]] == true )
14.             selection_probability[j]=0.0;
15.          else {
16.             selection_probability[j]=choice_info[c][nnList[c][j]];
17.             sum_probabilities = sum_probabilities + selection_probability[j];
18.          }
19.       }
20.       if (sum_probabilities==0)
21.          ChooseBestNext(step);
22.       else {
23.          r = rand.nextDouble() *  sum_probabilities;
24.          j=0;
25.          p = selection_probability[j];
26.          while (p<r) {
27.             j++;
28.             p = p+selection_probability[j];
29.          }
30.          tour[step] = nnList[c][j];
31.          index[nnList[c][j]]=step;
32.          visited[nnList[c][j]] =  true;
33.       }
34.    }
```

First we initialize some variables vital for the tour decision. Most noticeably we create an array named selection_probability, which includes the chance of going from a city c to every city j (where j is every city on c's feasible neighborhood) and sum_probabilities witch is the sum of their choice info. At line 11 to line 20 a for loop starts that runs for nn times, used to fill the Selection_probability array and calculate the sum_probabilities. If all the Nearest Neighbors have been visited we call chooseBestNext method, that inserts to the tour the city with the highest choice information. If on the other hand the sum_probabilities had a value higher than 0, meaning not all the cities of the feasible neighborhood was visited, a random number called r is computed with a value between 0 and sum_probabilities as can be seen in line 25. Then we check which city has selection probability closest to that random number and we add it to the tour array.

3.3.1: Ant Swarm Update Statistics:

at every iteration after the Ants constructed their tours the statistics of the program are updated:

```
1.  public void UpdateStatistics(int i)
2.     {
3.        for (int k=0; k<m; k++) {
4.           ants[k].cal_tour_length();
5.           if (LastbestTourLength>ants[k].getTourLength()) {
6.              eliteAnt=ants[k];
7.              LastbestTourLength=eliteAnt.getTourLength();
8.           }
9.        }
10.       if (bestTourLength>LastbestTourLength) {
11.          bestTourLength=LastbestTourLength;
12.          out.println("Best tour length="+bestTourLength+" after: "+(i+1)+" gap =
     "+((double)(bestTourLength-optimalTour)*100.0)/(double)optimalTour+"%");
13.          eliteAnt.draw(x,y,0);
14.       }
15.       LastbestTourLength=Integer.MAX_VALUE;
16.    }
```

First a for loop find the best iteration tour length and saves it as LastbestTourLength (line 4-10), if this is the best so far tour length we save it as bestTourLength and we print it next to a number called gap describing how far away we are from the optimum and then print the tour map. The way that we calculate the optimum is:

Function 3.1: gap from optimum $= ((bestTourLength - OptimalTourLength) * 100)/ OptimalTourLength$

Finally the visualization of the tour is printed out in line 21. To print out the visualization of the tour we have used two java classes named drawTour and stdDraw made by our supervisor Keld Helsgaun.

3.3.2: Ant Swarm Pheromone Update Method:

```
1.  public void ASPheromoneUpdate()
2.  {
```

```
3.  Evaporate();
4.        for (int k=0; k<m; k++)
5.           DepositPheromone(k);
6.  }
```

Hereafter the pheromone matrix is updated by calling the ASPheromoneUpdate method. First the ASPheromone calls the evaporate method in line 3 that evaporates pheromone on each edge between the cities by decreasing their values according to an evaporation value p.

```
1.    public void Evaporate()
2.    {
3.       for (int i=0; i<n; i++)
4.          for (int j=i+1; j<n; j++) {
5.             pheromone[i][j] = (1.0-p) * pheromone[i][j];
6.             pheromone[j][i] = pheromone[i][j];
7.          }
8.    }
```

From line 3-6 we evaporate pheromone on every edge between every city. To reduce the computing time we set pheromone[j][i] equal to pheromone[i][j] because the pheromone matrix is symmetric, for example the edge between city 5 and 1 is the same as the edge between 1 and 5. The evaporation could be smaller or bigger but it is suggested by Dorigo to evaporate it by half (DS, 2004: 74).

Next in line 4-6 in the ASpheromoneUpdate the depositPheromone method is called for each ant.

```
1.    public void DepositPheromone(int k)
2.    {
3.       int j,l;
4.       double trail = 1.0/(double)ants[k].getTourLength();
5.
6.       for (int i=0; i<n; i++) {
7.          j = ants[k].getTourCity(i);
8.          l = ants[k].getTourCity(i+1);
9.          pheromone[j][l] = pheromone[j][l]+trail;
10.         pheromone[l][j] = pheromone[j][l];
11.      }
```

12. }

In the DepositPheromone at line 4 the pheromone that an ant deposit is set equal to $1/T^k$ where T^k is the size of the tour the ant have found. This means that the ants lay down a stronger pheromone trail the smaller a tour they found. In line 6-10 the pheromones are deposited on every edge that the ant used to create its tour. The way the pheromone is calculated can be seen in function 2.2.

The next action done in the loop in control is to update all the choice information after the evaporation and deposit of pheromones have been performed. The for loop described now in the control keeps running for a set period of iterations and when it finishes the program ends.

4: AS code variations

4.1: EAS

The first improvement to the AS algorithm is the EAS. The main difference between AS and EAS is that in EAS there is added extra pheromone enforcement to the edges of the iteration best tour.

4.1.1: EAS

To implement the EAS improvement only a few small changes are needed to the code. First we have changed our initiation of our pheromones to an equation suggested by Dorigo(DS, 2004: p. 71).

Function 4.1: $pheromones[i][j] = (e + m)\ /\rho * nntl$

Where e is a parameter which describes how much weight should be given to the iteration best tour. We have also added an extra update pheromone rule called EliteDeposit which is only

executed for the iteration best so far ant.This pheromone update is e*Δt_{ij}^{bs} and is only added to the edges of the iteration best tour so the formula is:

4.2 Function: $\Delta t_{ij}^{bs} = \begin{cases} \frac{1}{C^{bs}} if\ arc\ i\ j\ belongs\ to\ T^{bs}; \\ 0 \quad Otherwise \end{cases}$

Where T^{bs} is the iteration best tour and C^{bs} is its length. So the function 2.7 for the pheromone deposit after each iteration now becomes (DS, 2004: p. 73).

4.3 Function: $t_{ij} = t_{ij} + \sum_{k=1}^{m} \Delta t_{ij}^{k} + e\Delta t_{ij}^{bs}$

By implementing the EliteAnts improvements the tours goes both closer to the optimum and finds a good tour in less iterations than the AS.

4.1.2: EAS implementation

We will now show the main changes that we have done to our code in order to implement EAS. First we create a new ANT object called eliteAnt at the updateStatistics method as above, which is the ant that found the best tour length at every iteration. This is the ant that will deposit the extra amount of pheromone. So at every ASPheromoneUpdate after the normal evaporate and deposit the EliteDeposit is called.

```
1.  public void EliteDeposit()
2.  {
3.     int i, j, l;
4.     double trail = (double)e/(double)eliteAnt.getTourLength();
5.     for (i=0; i<n; i++) {
6.        j = eliteAnt.getTourCity(i);
7.        l = eliteAnt.getTourCity(i+1);
8.        pheromone[j][l] = pheromone[j][l]+trail;
```

```
9.              pheromone[l][j] = pheromone[j][l];
10.         }
11.     }
```

As can be seen in line 4 the trail in the eliteDeposit is calculated as in formula 4.3. Hereafter we get the edges of the iteration best tour of the eliteAnt and we deposit the pheromone as before shown in DepositPheromone in line 5-9.

4.2: Min Max AS

4.2.1: Min Max AS Behavior

The other extension we have implemented to our AS is the Min Max As. The min max AS variation can be considered the best at finding a tour as close to the optimal, amongst the standard AS variations on most instances though closely followed by ACS (DS, 2004: p. 92). ACS is by many favored over Min Max though, because unlike Min Max AS it finds a very good tour in a shorter amount of time. Actually Min Max has the worst performance for the first iterations but it first stagnates when coming very close to the optimum which leads to that it outperforms the other AS variations after enough iterations (DS, 2004: p. 93).

4.2.2: Min Max Pheromone deposit:

In the min max there is only one ant that deposit pheromone which is either the iteration best ant or the best so far ant. The deposit is done either on the best tour found in an iteration if for the iteration best ant or the best tour found since the start of the program by the best so far ant. So the pheromone deposit becomes.

Function 4.4: $t_{ij} \leftarrow t_{ij} + \Delta e_{ij}^{best}$

Where Δe_{ij}^{best} is either iteration best ant our best so far ant. In our program we have chosen to make an extension so both are used to have the optimal performance of Min Max AS. Whether it is the iteration best or the best so far ant that deposit pheromone is decided at random so 9 out of

10 times the pheromone deposit is by the best so far ant, otherwise the iteration best ant deposit. So the equation for the pheromone deposit becomes.

4.4 Function: $t_{ij}^{best} = \begin{cases} 1/C^{9b}, & if\ q < q_0; \\ 1/C^{ib}, & Otherwise; \end{cases}$

Here q is the random number between 0-1 and q_0 is a parameter in the program J in the ASdecisionrule as described in 3.2.1 which we has set to 0.9 as suggested in Dorigo(DS, 2004: 71). C^{9b} is here the best so far tour length found where C^{ib} is the iteration best tour length.

4.2.3: Min Max AS pheromone limits

As can be noted that if we only allow the iteration best ant or the best so far ant to deposit pheromone we will get a extremely biased program with almost no exploration at all. We have therefore made 3 changes to our AS so the ants doesn't become biased. First we have set a max and minimum value for our pheromones values. The max and min change as the program goes on. It is required that throughout the program that the min and max values follow this rule:

4.5 Function: $0 < tmin < t_{ij} < tmax$

To secure that this rule is upheld we have chosen to calculate the tmin and tmax as suggested by Dorigo(DS, 2004: 71). So we set the tmin and tmax to:

4.6 Function: $tmax = 1/(p * C^{9b})$

4.7 Function: $tmin = tmax(1 - \sqrt[n]{0.05})/((avg - 1) * \sqrt[n]{0.05})$

Where p is the pheromone evaporation, n is the number of cities of the instance and avg is the average number of choices an ant has when choosing its next city which equals to n/2. It may be hard to see the logic in why the tmax and tmin should be calculated by these exact formulas, but it

has been proved that the tmax and tmin should be set to this. The proof can be found in *Max Min Ant System* by Thomas Stützle and Holger H. Hoos (SH, 2000: pp 15-17).

4.2.4: Min Max AS Pheromone initialization and reinitialization

The next change to our AS is that we initialize our pheromones to an approximate of the upper pheromone value(tmax). Our pheromone initialization is therefore:

4.8 Function: $t_{ij0} = 1/(\rho * nntl)$

As can be seen the pheromone initialization use the same formula as for calculating tmax except that we use the Nearest Neighbor Tour Length as the best so far tour. Last thing we do to prevent stagnation is that after a number of iterations where there haven't been found a new best tour, we reinitialize the pheromones to the upper limit of the pheromones (DS, 2004: p 76). We have in our AS chosen to reinitialize the pheromones after 60 iterations where no better tour has been found.

By applying these changes we get an ACO which never become biased towards a route and keeps trying to improve upon the best tour found since the start of the program. The result is an ACO which initially finds very bad tours, but keeps finding relevant improvements until it goes very close to the optimum.

4.2.5: Min Max Implementation

We will here explain the main changes we have made to our program to implement the Min Max implementation. The main changes has been within UpdateStatistics, creating a new class called MMDeposit and MMEvaporate. First we have made changes to the UpdateStatistics used for the EAS extension to make it update needed statistics for the MMAS.

```
1.  /* as in UpdateStatistics for EAS
2.        if (bestTourLength>LastbestTourLength) {
3.           bestTourLength=LastbestTourLength;
4.           if (algorithm.equals("m")) {
5.              for (int j=0; j<n+1; j++)
6.                 bestTour[j]=eliteAnt.getTourCity(j);
7.              tmax=1.0/(p*bestTourLength);
```

```
8.              tmin= tmax*( Math.pow(0.05, 1.0/n))/(((n/2)-1)*Math.pow(0.05, 1.0/n));
9.              IterationCount=0;
10.         }
11. /*line 11-12 as in UpdateStatistics
12.
13.  }
14.       else if (algorithm.equals("m")) {
15.          IterationCount++;
16.          if (IterationCount>60) {
17.             IterationCount=0;
18.             InitializePheromoneTrail();
19.          }
20.       }
```

In the if statement at line 2, where we check whether we have found a better tour, we update the statics. In line 4 we specify that these statistics should only be updated for the MMAS. We then save the whole tour array of the best tour found so far in line 5-6. In line 7-8 we recalculate the max and minimum as described in formula 4.6 and 4.7. Last in line 9 we reset a variable called iterationCount to 0. We have added a else if sentence which has the condition that MMAS is activated, to check when at each iteration we don't find a better tour. Within the else if sentence we increment the variable IterationCount by 1 everytime it is called in line 15. We then check in line 16 if a better tour hasn't been found for 60 iterations. If this is the case the iterationCount is reset and all pheromones are reinitialized to tmax to restart the explorative phase.

Next we have made a new pheromone deposit for MMAS that deposit pheromone either on the edges of the bestTour or the iterationBest tour.

```
1.  public void MMDeposit()
2.     {
3.        int i, j, l;
4.        double q;
5.        q=rand.nextDouble();
6.        if (q0<q) {
7.           double trail = (double)1/(double)eliteAnt.getTourLength();
8.           for (i=0; i<n; i++) {
9.              j = eliteAnt.getTourCity(i);
10.             l = eliteAnt.getTourCity(i+1);
11.             pheromone[j][l] = pheromone[j][l]+trail;
12.             if (pheromone[j][l]>tmax)
13.                pheromone[j][l]=tmax;
14.             pheromone[l][j] = pheromone[j][l];
```

```
15.          }
16.       }
17.       else {
18.          double trail = (double)1/(double)bestTourLength;
19.          for (i=0; i<n; i++) {
20.             j = bestTour[i];
21.             l = bestTour[i+1];
22.             pheromone[j][l] = pheromone[j][l]+trail;
23.             if (pheromone[j][l]>tmax)
24.                pheromone[j][l]=tmax;
25.             pheromone[l][j] = pheromone[j][l];
26.          }
27.       }
28.    }
```

First we create a new variable q which is set to a random value between 0-1 in line 4-5. In line 6 we compare this variable with a parameter called q0 which is initiated to 0.9, so the requirement for the if sentence will be fulfilled 1 out 10 times. If it goes in the if sentence the pheromones are deposited on the iteration best tour, using the tourLength of the iteration best ant to calculate the amount of pheromone to deposit. We check for each deposit on an edge whether the pheromone value of the edge between city I and j has become higher than the tmax and if so we set the amount of pheromone on the edge to tmax at line 8-15. We do the exact same check at line 22-24. If it doesn't go in the if sentence it instead deposit pheromones on the edges of the best tour so far, using the length of the best so far tour to calculate the amount of pheromone to deposit at line 18-21.

The last method method MMEvaporate has only one change from the normal evaporate.

```
1.    public void MMEvaporate()
2.    {
3.  /* same as the Evaporate method.
4.             if (pheromone[i][j]<tmin) {
5.                pheromone[i][j]=tmin;
6.             }
7.  /* same as the Evaporate method.
```

As can be seen from this line of code the evaporate method now check every time it evaporates pheromone on an edge, whether the pheromone on the edge has become lower than the minimum value.

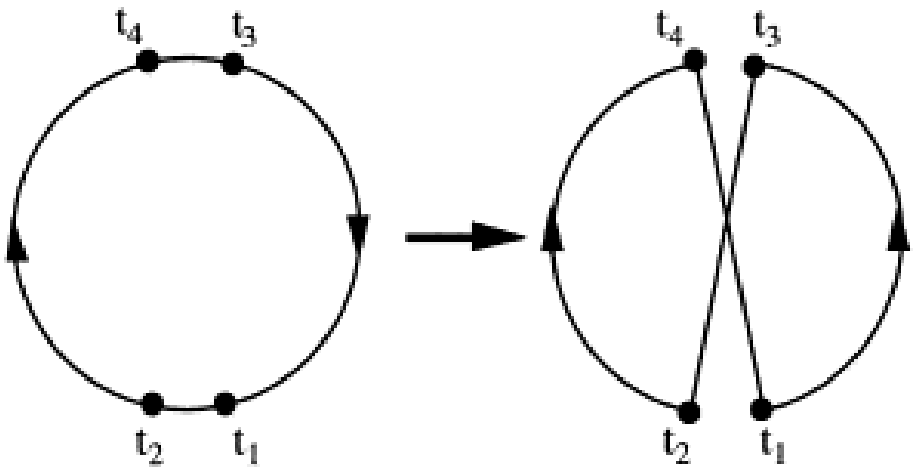
5: Non ACO Improvements

5.1: local search

We have chosen to use a local search algorithm within our program to improve the tours we find. A local search algorithm within an ACO checks whether there can be made any improvements to the tour of each ant after every iteration. We do this by repeatedly checking whether different combinations of the tour yields a smaller tour length. The local search algorithms vary in the way they combine different cities in the tour and examples of them are 2-optimization(2-opt) and 3-optimization(3-opt). In our program we use 2-opt and extend it to also perform an extra move called 2.5-opt (DS, 2004: p. 92).

5.2: 2-opt theory:

The 2-optimization algorithm try new combinations of a tour, by repeatedly checking if removing two edges in the tour and replacing them with two new ones produces a better tour. A visualization of a 2-opt move is shown below:



Picture taken from (K, 2000: chapter 2.2.2).

What can be seen from the left part of the figure is that before the 2-opt move we have two edges between city a to city b and city d to city c. On the right part of the figure we can see that the edge

between a to b and d to c has been replaced with two new edges between a to d and b to c. To remove the old edges and create the new ones we swap city b with d (K, 2000: chapter 2.2.2). It can also be noted that the order of the cities between b and d is reversed signified by change of the direction arrow. An example of a full 2-opt move could be:

$$\begin{array}{rccccccccc} Before\ reversing\ before\ 2-opt\ move: & 9a & \mathbf{1b} & 2 & 5 & 4 & 3 & 6 & \mathbf{7d} & 8c \\ Before\ reversing\ after\ 2-opt\ move: & 9a & \mathbf{7d} & 2 & 5 & 4 & 3 & 6 & \mathbf{1b} & 8c \\ After\ reversing\ after\ 2-opt\ move: & 9a & \mathbf{7d} & 6 & 3 & 4 & 5 & 2 & \mathbf{1b} & \mathbf{8c} \end{array}$$

Example of a tour Array before and after a 2-opt move.

In the example above the two cities that we swap in the 2-opt move is city 1 and 7, answering to city b and d. City a is here city 9 and city 8 is c. At the first line there is an edge between 9-1 and 7-8 answering to an edge between a to b and d to c. At the second line city 7 and 1 have been swapped thus creating an edge between a to d and b to c. It can be noted here that before the reversing we have created a new edge between 7 to 2 and 6 to 1 that we don't want to create when making a 2-opt move. We therefore reverse the order of the tour between b to d. By doing this it can be seen that every edge is the same as before, considering the fact that tours edges are symmetric, except for the two edges we wanted to create between a to d and b to c. So after performing these actions we have performed a 2-opt move.

Obviously not every move in the 2-opt will yield a better tour. It is therefore important to check whether any gain in tour length is found by replacing edge a to b and d to c with a to d and b to c. The formula for this is:

Formula 5.1: $Gain = Distance[a][b] + Distance[d][c] - (Distance[a][d] + Distance[b][c])$

Here gain is the amount of tour length improvement found by making the 2-opt move. If gain has any positive value it means that we found a gainful move and should start changing the tour array as described above (VAS95, 1995: 177). The algorithm stop when no other 2-opt moves can be found.

In our code every edge is tested for a possible 2-opt move. When every possible 2-opt move has been tried the program checks for whether any gain was found during the 2-opt moves. If yes the 2-opt continues, trying to improve the newly improved tour once more. If after trying all possible

2-opt move for every edge and no gain is found the 2-opt terminates. The 2-opt is performed for every ant in the programs tour and when all tours have been improved as much as possible by the 2-opt, the 2-opt method terminates and the program continues.

By using the local search we manage to find a much better tour by several percent closer to the optimum. Below there is an example of a visualization of an instance before and after 2-opt has been applied.

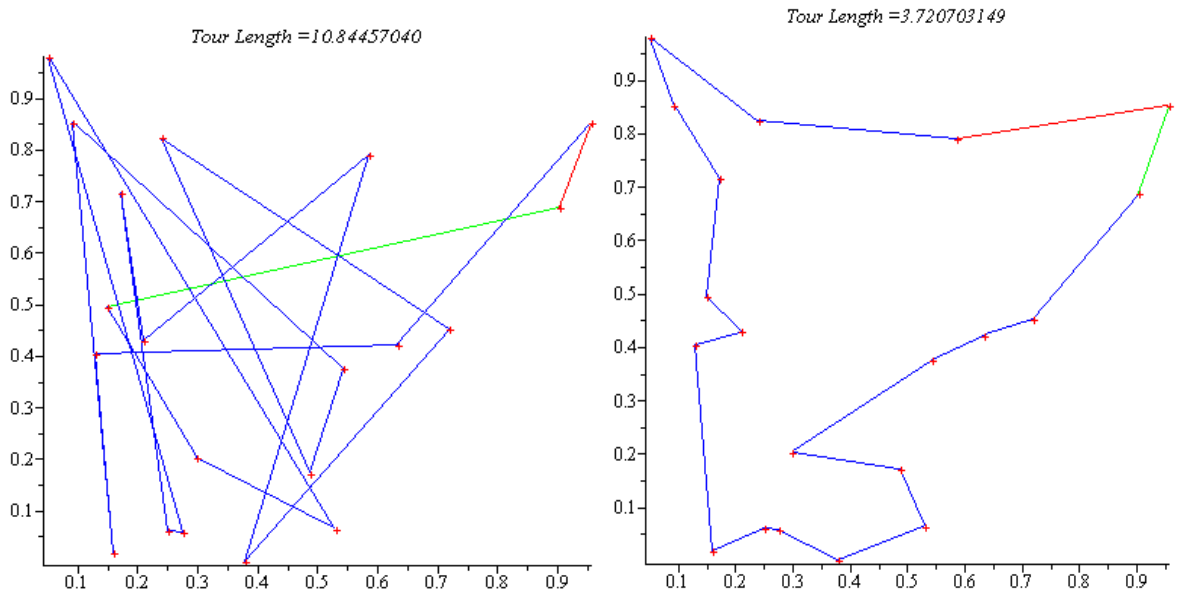


Figure 5.1 found at google pictures (see Bibliography for links).

5.2.2: 2-opt implementation

We will now describe how we have implemented 2-opt and its improvements (dontLookBit and 2,5 opt)

```
1.  public void Opt2()
2.  {
3.      double maxGain;
4.      double gain;
5.      boolean isChange;
```

```
6.      boolean[] dontLookBit = new boolean[n+1];
7.      for (int j=0; j<=n; j++)
8.         dontLookBit[j]=false;
9.      do {
10.     isChange=false;
11.     for (int i=1; i<n; i++) {  //for every city try 2-opt
12.        if (dontLookBit[i-1]==false) {
13.           int a = tour[i-1];
14.           int b = tour[i];
15.           int c,d;
16.           maxGain = 0;
17.           gain = 0;
18.           int maxEdge = 0;
19.           for (int j=1; j<=n; j++) {
20.              if ( (j!=i) && ((j-1)!=i) && ((j-1) != (i-1)) && (j != (i-1)))
21.              {
22.                 c = tour[j];
23.                 d = tour[j-1];
24.                 //  calculate if there is gain
25.                 gain = ( distance[a][b] + distance[c][d] ) - ( distance[a][d] + distance[b][c] );
26.                 if (gain > maxGain) {    // find the most gainful move
27.                    maxGain = gain;
28.                    maxEdge = j-1;
29.                 }
30.              } //end if
31.           } //end for j
32.           if (maxGain > 0 ) {     // if there is gain make the 2-opt move (flip 2 cities)
33.              int u;
34.              int v;
35.              for ( u = maxEdge, v = i; u>=i;  u--, v++)
36.              {
37.                 if (v < u) {       //  change also the order of the cities (u--)
38.                    flip(v, u);
39.                 }
40.              }
41.              isChange = true;
42.              dontLookBit[i-1]=false;
43.              dontLookBit[i]=false;
44.              dontLookBit[maxEdge]=false;
45.              dontLookBit[maxEdge+1]=false;
46.           } //end if
47.           else {
48.              dontLookBit[i-1]=true;
49.           }
50.        }
```

```
51.       }// end for i
52.    }while (isChange==true);
53.    }
```

The 2-opt move starts with a do-while loop (line 9) that runs until every possible 2-opt has been performed. Inside the loop there is two for loops used for checking all possible 2-opt moves for all the edges in a tour array. In lines 11-14 we set city b to position i in the tour array and a to the previous of b in the tour array. In line 19 the second loop starts within our first loop that runs until a 2-opt move has been tried on every edge c,d with the current edge a,b. In line 20 there is a condition for performing a 2-opt move, that the cities a and b isn't the same as c and d. at line 21-22 we set c to position j in the tour array and d to the previous of c. At line 24-25 we check whether any gain is found by replacing the edges a,b and c,d with a,d and b,c. If the gain has a positive value that is lower than the maxGain, that has been found for edge a-b we save the position of city b in the tour and set the variable maxGain to gain at line 26-27. When the for loop j has finished and the most gainful move has been found we flip the position of b and d along with all the cities between city b and city d at line 34-38.

5.3: -opt improvements

5.3.1: dontLookBits

There are several ways to improve the running time of the 2-opt at the cost of producing slightly worse tours so the 2-opt performed isn't 2-optimal. We have therefore chosen to implement the speed up of the 2-opt by using the don't look bits a technique first introduced by Bentley at 1992. What you do is that you associate a binary (true, false) bit for every city in the tour. At first all the bits are initialized to false. Every time there is no 2-opt move found for an edge in the tour array the dontLookBit of the city of that edge that comes first in the tour (in our case city a) is set to true. By doing this, the 2-opt algorithm doesn't check for improvements in edges that didn't lead

to a 2-opt move before. Bentley also suggests that when an improvement is found the dontLookBits of all the cities that was involved is set to false (JM, 1995: pp. 26-27).

5.3.2: 2,5 optimization

For further improvements after we optimize the tours with 2 opt we perform to them a further optimization called 2.5-opt. In the 2.5-opt instead of removing and adding two edges, we move a city between two other cities in the row (C,D,E). We here remove the city in between (D) and add it between two other cities (A,B) to decreases the tour length(JM, 1995: p. 32). So at each try for a 2.5-opt move to calculate if there is gain the formula is:

Formula 5.2: $Gain = Distance[a][b] + Distance[b][e] + Distance[c][d] - (Distance[a][e] + Distance[d][b] + Distance[b][c])$

If there is Gain:

1. the D is saved as a temporary value
2. all cities between city C and B are moved one position back in the tour array. (from C to B → tour[i-1]=tour[i])
3. The temporary value is added to the position where city A was.

So before 2,5 opt move the tour is: C **D** E A B and after it is: C E A **D** B.

5.3.3: 2,5 implementation

To implement 2,5 opt the main change to our program is that we have made a move method that is called when a gain is found. Instead of calling the method flip() now we call move() that makes the necessary flips. The move method is:

```
1.  public void move (int bPos, int dPos) {
2.      int temp=tour[bPos];
3.      for(int i=bPos; i<dPos; i++) {
```

```
4.              tour[i]=tour[i+1];
5.          }
6.          tour[dPos]=temp;
7.      }
```

In line 2 we save the city we are going to move in a temporary variable. Then we copy all the cities between the position of the city b and the city d as explained in the theory, one place before their original position in the tour array at line 3-4. Then we copy city b to city d's old position in line 6.

5.3 the use of THREADS

Looking at our program code it can be reasoned that where the most cpu time is spent is on the contsruction and improvement of each ants tour. One way to decrease that running time is by the use of threads. We create a new thread for each ant, that contructs a new tour, improves it with 2-opt and 2,5 opt, and then it dies. So the methods which we run as threads are the NeighborListDecisionRule, 2-opt and 2.5-opt which are 3 of the most time consuming methods. by implementing threads ants can run in parallel so all the cores of the computer are used. The speed up depends on the number of cores the cpu has, so for the tour contruction and local search improvement the run time becomes: new run time=old run time / number of cores.

6: User Guide

All the compiled java class files of the program are compressed in a jar file named AsTSP.jar so anybody can execute it by double clicking it or by command line by typing: java -jar AsTSPjApp.jar.

To start finding solutions some parameters should be given by the user by typing -'parameter name''parameter value'. The list of them appears by executing the program with the parameter -h (help). First of all the user must give a file name that contains all the coordinates of the cities and the number of them by typing -ifilename. Hereafter the user should give the optimal tour, if it is known so the application can compare its results by printing the gap between the tours found and the optimal tour (-o'optimum tour length'). Furthermore the parameter controlling the number of ants (-m'number of ants') needs to be set.

Other parameter are:

→ The algorithm that the program will use (-c), which can be 'a' for simple AS, 'e' for EAS or 'm' for MMAS.

→ -e factor witch declares how much weight is given to the best tour found in the elite ant algorithm.

→ -l which can be 0 or 1 and presents activation or deactivation of local search in the founded tours.

→ -a weight of pheromones factor.

→ -b weight of heuristic factor.

→ -nn number of nearest neighbors of each city.

→ -t number of times (iterations) the algorithms are going to run, if the optimal tour is not found.

→ -p witch Is the evaporations factor, that declares witch amount of pheromones will evaporate after every iteration. It can be noted that for the MMAS the p automatically is set to 0.02.

→ -q0 The factor that decides whether iteration best ant or best so far ant deposit pheromones in MMAS.

The parameters can be given in random order and if some of them are missing the programs runs with the default parameters: -id198.tsp -m50 -ce -l1 -a1 -b5 -nn20 -o0 -t1000 -e=50 -p0.5 -q0.9

7: Benchmarking

The benchmarking was performed on a Intel(R) Core(TM) i5 CPU (4 cores) 2.53GHZ with 4GB RAM. We will in this part benchmark our AS, EAS extension and MMAS extension, on a number of selected instances. When benchmarking we will measure the run time of the program, number of iterations needed to find the best tour, best tour length and gap from the optimum. We have chosen to benchmark our AS and EAS for 1.000 iterations and our MMAS on 5.000 iterations. The reason for this is that no significantly better tours are found after 1.000 iterations while MMAS

need to run for about 5.000 iterations before its improvements can actually be seen. We will at our graph be showing AS, EAS and MMAS development without 2.5-opt for 5.000 iterations to show that hardly any improvements occur after that many iterations. For each benchmarking of an ACO extension without 2-opt we will get the average of time, iterations to find best tour, best tour length and gap from optimum. We therefore run three tests for each ACO extension and get the average. The different benchmarks we will perform is:

- Benchmarking for AS, EAS and MMAS on 3 number of instances getting the average of 3 test runs without using 2-opt. Showing a graph with the development of AS, EAS and MMAS after 5.000.
- Benchmarking for AS, EAS and MMAS on 3 number of instances with 2-opt showing AS and EAS development for 1.000 iterations and MMAS for 5.000 iterations.
- Benchmarking our EAS with and without threads to show the difference in running time.
- Benchmark 2-opt with and without the 2.5-opt extension.

The instances which we will be benchmarking on is d198, a280 and rat783. As we benchmark we will compare our result to Marco Dorigo's and Thomas Stützles results from the text Ant Colony Optimization and see how well they match. Our benchmarking schema used for our first two benchmarks is build up like:

Name of benchmarking													
Instance	**Optimum**	**AS**				**EAS**				**MMAS**			
		time	**IT**	**TourL**	**Gap**	**Time**	**IT**	**TourL**	**Gap**	**time**	**IT**	**TourL**	**Gap**
d198.tsp	15780	1.000	999	50.000	99.99%	1.000	999	50.000	99.99%	1.000	999	50.000	99.99%
a280.tsp	2579	1.000	999	50.000	99.99%	1000	999	50.000	99.99%	1.000	999	50.000	99.99%
rat783.tsp	11340	1.000	999	50.000	99.99%	1000	999	50.000	99.99%	1.000	999	50.000	99.99%

Here IT is the average number of iterations it took to find the best tour, time is the how long it took for the program to finalize, TourL is the tour length of the tour found and Gap is the gap from the optimum.

7.1: Benchmarking of AS, EAS and MMAS without 2.5-opt

Our benchmarking results for AS, EAS and MMAS without 2.5-opt has been done with an alpha and delta value of 1 and 5 and for the EAS the e is set equal to n. We have in general used the parameters suggested by M. Dorigo and T. Stützle in Ant Colony Optimization (DS, 2004: p. 71). AS is run with the suggested amount of n ants by M. Dorigo and T. Stützle for the first two instances except rat783 where the ants are set to 100 while EAS and MMAS is run with 100 ants on all instances to have a decent run time.

Benchmarking of AS, EAS and MMAS without 2.5-opt													
Instance	**Optimum**	**AS**				**EAS**				**MMAS**			
		Time	**IT**	**TourL**	**Gap**	**Time**	**IT**	**TourL**	**Gap**	**time**	**IT**	**TourL**	**Gap**
d198.tsp	15780	32,6	400	16.908	6,52%	28,6	686	16.270	3,31%	107	3138	16.132	2,23%
a280.tsp	2579	66,5	330	2.867	11,16%	38,4	546	2.650	2,78%	121*	3336*	2.593	0,56%
rat783.tsp	8810	197	272	10.354	17,53%	202,3	960	9.223	4,69 %	980	4938	9.029	2.49%

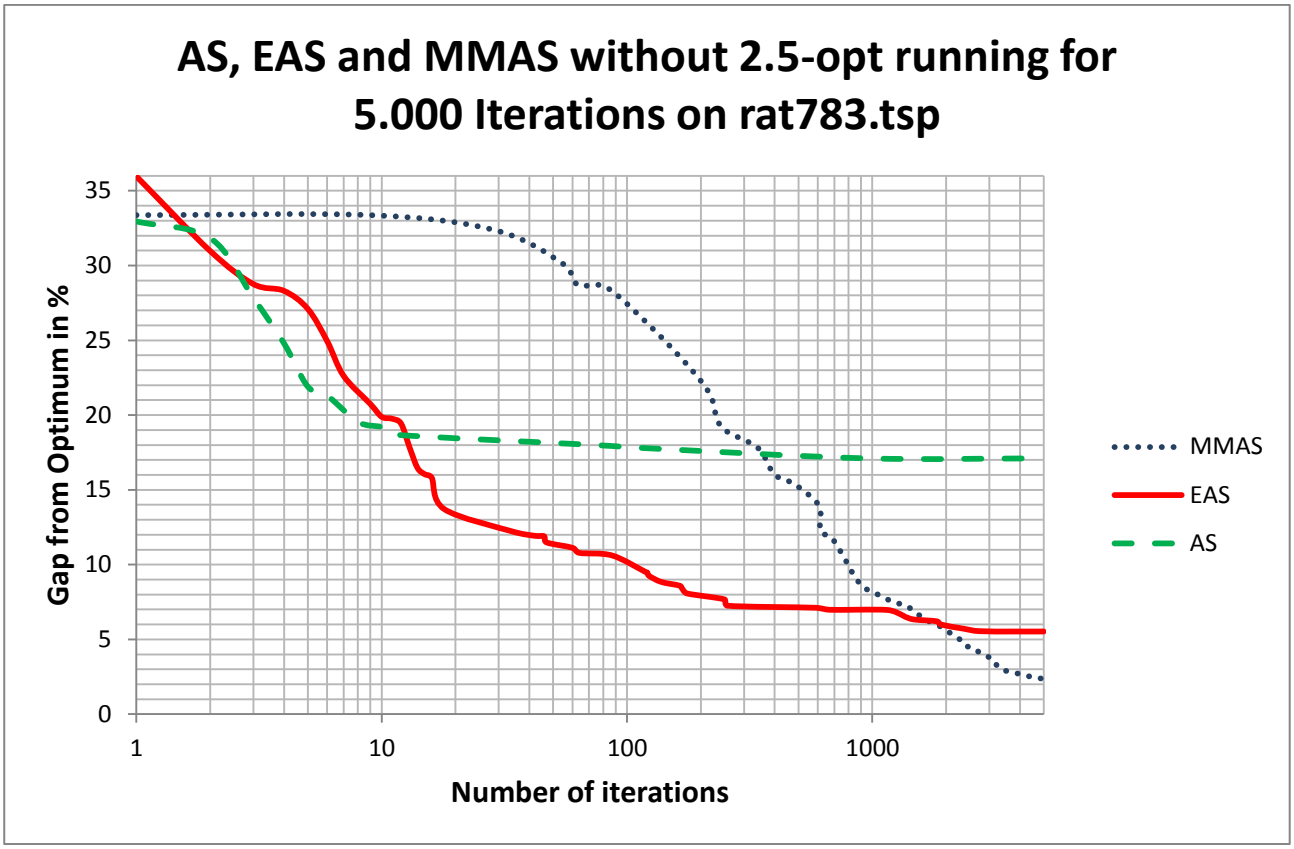
*found optimum one of the times

As can be seen we generally produce good results with every algorithm on the d198 with AS, EAS and MMAS. Compared to M. Dorigo and T. Stützle we have a slightly higher gap for AS and MMAS on d198.tsp, where he got around 6% with AS and 1.9% for MMAS. With EAS on the other hand we have a slightly better result than M. Dorigo and T. Stützle who where about 3.8% away from the optimum (DS, 2004: p. 92). The running time for AS compared to EAS is slightly higher because the ants are set to n instead of 100 for AS to slightly improve the tour construction results. It can be seen that AS doesn't handle the 280 city instance very well and produces a tour far away from the optimum. EAS on the other hand find a very good tour while MMAS finds the optimal once without the use of 2-opt and in average is under 1% away from the optimum. For the rat783 we found a slightly worse tours than M. Dorigo and T. Stützle for the AS and MMAS. This could be contributed to that we don't run both algorithms as long as M. Dorigo and T. Stützle do. EAS though surprisingly outperforms M. Dorigo and T. Stützle by several percent.

As can be seen the time needed to find a good tour is substantially increased for this instance especially for the MMAS because it runs for 5 times as many iterations. This is in accordance with M. Dorigo and T. Stützle's results where he had 100 times the running time for rat783 compared to the running time for d198 before every ACO algorithm stopped finding better tours. So it can be noted that the running time before finding as optimal a tour as possible with an ACO algorithm increases aggressively the larger the instance becomes. This is in accordance with that ACO algorithms mainly perform well on smaller instances. So even through some ACO algorithms perform well on larger instances, such as the MMAS, the running time starts going towards an unacceptable amount.

7.1.2: Graph of AS, EAS and MMAS without 2.5-opt

We will here show a graph of how AS, EAS and MMAS improve the tour the longer the program runs. Here the x-axis is the number of iterations the program has run and the y axis the gap from the optimum. We have run the test for 5.000 iterations on rat783.tsp without 2.5-opt. We have tried to recreate the same graph from *"Ant Colony Optimization"* by M. Dorigo and T. Stützle except that instead of having CPU time in seconds on the x axis we have number of iterations.



As can be seen from the graph AS has the best performance for 10 iterations closely followed by EAS while MMAS performs poorly in the start. After 100 iterations AS have almost stagnated while EAS is the closest to the optimum though the graph for EAS starts to even out compared to its start. MMAS is improving but still doing rather poorly. At 1.000 AS have completely stagnated while EAS still finds slightly better tours. MMAS has improved dramatically and have almost as good results as EAS. At around 3.000 iterations EAS starts to stagnates and is now out performed by MMAS. At 5.000 iterations we see that EAS has completely stagnated while MMAS is starting to show signs of stagnations after having found a tour at around 2,3% away from the optimum. So what can be concluded is that around 1.000 AS finds no more improvements, EAS has almost stagnated while MMAS has to run for about 5.000 iterations before finding a decent tour.

7.2: Benchmarking of AS, EAS and MMAS with 2.5-opt

While we did find reasonable result just using our three ACO algorithms we were still for most instances over 1% away from the optimum. We will therefore now try to see how good results we can produce using 2.5-opt to optimize the tour. It is suggested by M. Dorigo and T. Stützle to change certain parameters for the program when using local optimization because of the change of nature to ACO algorithms when applying 2.5-opt. We have by testing found that we finds a good tour as fast as possible by using 20 ants for every algorithm. The benchmark table for testing with 2-opt is:

Name of benchmarking													
Instance	**Optimum**	**AS**				**EAS**				**MMAS**			
		time	**IT**	**TourL**	**Gap**	**Time**	**IT**	**TourL**	**Gap**	**time**	**IT**	**TourL**	**Gap**
d198.tsp	15780	27	665	15.840	0.38%	28	787	15818	0.24%	24	885	15.780	0%
a280.tsp	2579	41	9	2.592	0.5%	3	23	2579	0%	23	500	2579	0%
rat783.tsp	11340	322	580	9174	4.13%	329	284	8872	0.7%	1488	3.167	8.813	0.03%

As can be seen from the table we get results for almost every test that are either less than 1% from the optimum or the optimal tour. Mainly MMAS provided the best results except for the a280.tsp where EAS with 2.5-opt also managed to find the optimal tour and in much less iteration. The only bad result is for AS with 2.5-opt on the rat783 instance on which we were more than 4% away from the optimum. This could be because AS doesn't put any weight on the good routes found so far compared to EAS and MMAS.

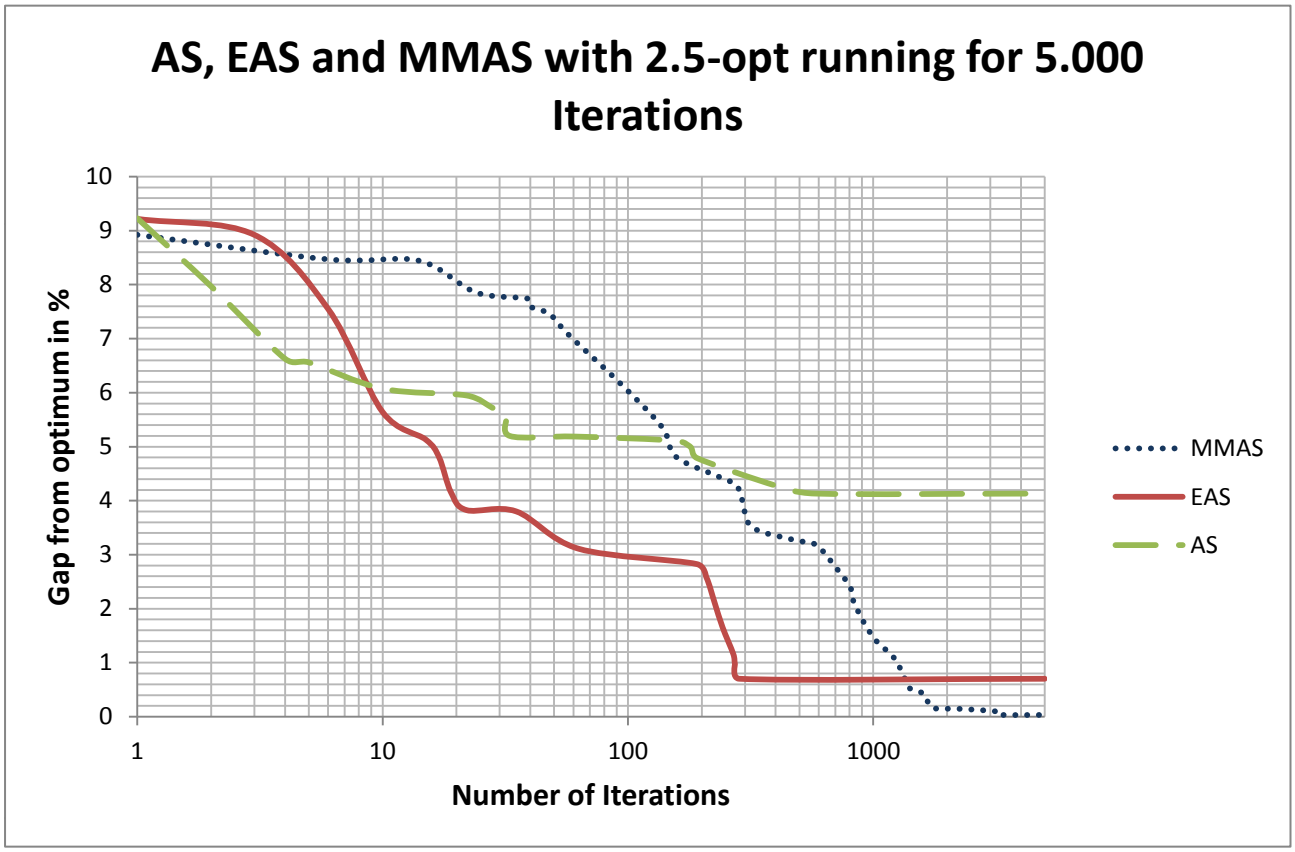
It can be noted that even though we only use 20 ants we still have a higher running time for the same number of iterations without 2-opt when using 100 ants. So while our 2.5-opt move is very effective at finding a good tour it has the drawback that it takes a lot of time to perform. If we compare the time it takes for MMAS to complete 5.000 iterations on rat783 with what it takes for it to complete the same number of iterations on the same instance with only 20

ants and with 2-opt we see that it takes 51,8% longer even though it used only 1/5 of the number of ants. So the 2-opt makes our program go about 8 times as slow as without it considering that:

$$\frac{100}{20} * 1.58 = 7.9$$

7.2.1: Benchmarking of AS, EAS and MMAS with 2.5-opt

We will here show a graph of a 2.5-opt benchmark done on rat783.tsp for 1.000 iterations for AS and EAS and 5.000 iterations for MMAS. The graph is done in the same way as our graph for AS, EAS and MMAS without 2.5-opt.



We see from the graph that AS and EAS stagnates before reaching 1.000 iterations. Especially EAS already stagnates after about 300 iterations at 0.55% from the optimum while AS first stagnates at

around 700 iterations. MMAS on the other hands first shows signs of stagnation after over 3.000 iterations where it finds a tour 0.03% away from the optimal tour.

7.2.2: Benchmarking of 2-opt with and without 2.5-opt extension

2.opt and 2.5 opt Benchmarking			
2-opt		**2.5-opt**	
Time	Gap	Time	Gap
26,33	0,25%	31,3	0,05%

As can be seen from this table the running time of 2-opt is slightly smaller than 2.5-opt and it looks like the program is about 18,7% slower with 2.5-opt considering that (31,3/26,33)*100 – 1 = 18,7%. This corresponds well with that a 2-opt move when extended with 2.5-opt is reduced in speed by 30-40%(JM, 1995: p. 32). On the other hand 2.5-opt serves its purpose of getting us slightly closer to the optimum of the tour and there can be seen a clear difference from the results of the two algorithms.

7.3 Benchmarking of threads

One of the most significant speed ups of our program is threads. Threads are one of the few improvements that creates no negative effects such as less optimal tours or higher running time that we have added to our program. We have ran EAS with 2-opt on d198 3 times both with threads activated and deactivated for 1.000 iterations to see the difference in speed. On the 4 core computer we used for testing the difference with and without thread in running time was:

Benchmarking with and without	
Time with Threads	**Time without Threads**
40,3	106,3

As can be seen we more than half our running time by using threads where The exact speed up is 163%. Ofcourse this speedup is completely computer dependent so the more cores the computer have the better speedup. It can be noted that the threads are only applied to the tour construction which is the part of our program that takes up the most time.

8: Discussion

After performing the benchmarking we see one central pattern for every benchmark which is that our program finds very good results in general but have a high running time. This can be attributed mainly to the extensions we chose to implement and a only sub optimized 2.5-opt algorithm that could in theory be running faster than it does at the moment. The implementation of MMAS especially makes our program output great results without 2.5-opt and with 2.5-opt it seems to be a very effective algorithm for finding the optimal solution. On the other hand if we had chosen to implement the ACS instead we should theoretically have nearly as good results and be able to find them faster considering that ACS always uses only 10 ants, performs well on big instances and uses only O(n) time on recalculating the pheromones unlike the other extensions that uses $O(n^2)$(DS, 2004: p 77). While we have a high running time for our ACO algorithms without 2.5-opt, it can be mentioned that the time results we get are not unexpected compared to Dorigo's results. So it could be pointed out that when implementing MMAS instead of ACS we chose to make an ACO algorithm where the main focus was finding a very good tour where if we had made ACS our goal would have been making a good tour finding algorithm with a good run time performance.

The 2.5-opt on the other hand could be speeded more up than it is now which is the main hindrance to our program at the moment. Considering that our best tour finding algorithm, MMAS with 2.5-opt, has an extremely long running time it means that if we had a faster 2.5-opt we would be able to find the best tour possible that our program can produce on larger instances within an acceptable time frame.

9: Conclusion

We will here conclude if what have implemented fulfill the requirement specifications of the start of our program. The two main goals that a good TSP solving algorithm should fulfill is as stated in the requirement specification:

- Have as good a solution as possible for the instance.
- Be able to solve a TSP problem as fast as possible.

We feel that our algorithm fulfill the first goal as well as could be expected for the extensions we implemented especially considering that we are able to find the optimal on many instances ranging from about 300 and less within a reasonable amount of time with MMAS with 2.5-opt. We have also shown that on larger instances, such as rat783.tsp, the MMAS with 2.5-opt can come under 0,1% away from the optimum. Considering our initial expectations for our program this goes beyond our initial requirements of our program to fulfill the first TSP requirement.

We don't fulfill the other main requirement for TSP, Solving instances as fast as possible, with our program, mainly because:

- We chose to focus on implementing the best tour constructing algorithm amongst the standard ACO algorithms(AS, EAS, ASrank, MMAS and ACS) while if we had implemented ACS we would have had better running time.
- Our 2.5-opt solution speed could still be improved.

So a focus on producing best possible tours over speed and a only semi-optimized 2.5-opt means that our program is lacking a bit at fulfilling the second requirement within expectations. We have though provided many speed up improvements in our program by including threads, nn_list for ants, precalculated matrices and dontLookBits for 2-opt. So while our program doesn't fulfill this requirement optimally it can't be said that we haven't got a program where the running time is taken into consideration.

In conclusion we have fulfilled all our requirement specifications, both TSP and non-TSP related, for our program except for getting as fast a 2.5-opt as possible.

10: Perspectivation

We will in this program reflect over how our program could still be improved beyond those improvements we already have, these improvements will be on the speed, tour finding and non-tsp related features. Concerning speed the first and most radical improvement that could be added is within the 2.5-opt by also using nnLists for it as we do with the ants. So for when performing 2.5-opt on a edge a,b we would only try to perform it for the nn nearest neighbors. This would provide a major speedup for the 2.5-opt and the program in general in run time, even though it would take more iterations to find the best tour possible. Also having implemented ACS and as a fourth runnable ACO algorithm, to find a good tour with a good running time, would have been another way to make a program where you can chose to have a better running time at the cost of slightly worse tours.

Of non-related ACO improvements, the possibility to include a GUI from where the program can be run and the parameters can be changed, could be an option. This would increase the usability of our program especially for people with little or no knowledge about java programming. We could also drop the drawTour and stddraw classes by Keld Helsaugn and make our own so our program relies only on our own code to work.

While we already got in our opinion a good tour finding algorithms there are still ways to improve the results. One is mentioned in (name of report of MMAS) called pheromone smoothing which is a minor improvement(about 0,2% closer to the optimum) for the ACO. It can be applied to all ACO extensions except for ACS. Another improvement, that would come at the cost of running time though, is using the 3-opt as a local search algorithm instead of only 2.5-opt. This would make the tour construction algorithm find significantly better tours but the time the local search takes would increase from $O(n^2)$ to $O(n^3)$(DS, 2004: p 94). If we truly wanted to optimize our local search we should apply the Lin-Kerrigan-opt but it is our understanding that doing so requires an implementation part beyond what is expected of a bachelor project.

11: Bibliography

[DS04] Dorigo M., Stützle T.(2004): *"Ant Colony Optimization"*, Massachusetts Institute of Technology, the MIT Press.

[BTD99] Bonabeau E., Theraulaz G., Dorigo M.(1999) *"Swarm intelligence: From natural to Artificial Systems"* Oxford University Press, Cary, NC USA.

[DB06] Dorigo M., Birattari M., Found at 24/05/11, "*Swarm Intelligence",* Scholarpedia http://www.scholarpedia.org/article/Swarm_intelligence

[GHL94] Gendreau M., Hertz A., Laporte G.,(1994): "*New Insertion and Postoptimization Procedures for the Travelling Salesman Problem"* INFORMS

[P94] Penna T. J. P.,(1994): "*Travelling salesman problem and tsallis statistics"* published in "*Physical Review E Statistical Physics, Plasmas, Fluids, and Related Interdisciplinary Topics – Third Series, Volume 51, Number 1 – Rapid Communcations Section"*, American Physical Society, United states.

[K00] – Helsgaun K.,(2000) *"An effective implementation of the Lin-Kernighan traveling salesman heuristic"* published in *"European Journal of Operational Research"*

[OP96] Osman I. H., Kelly J. P.,(1996) "*Meta-Heuristics: Theory & Applications"*, Kluwer Academic Publishers, United State.

[VAS95] Verhoeven M.G.A., Aarts E.H.L., Swinkels P.C.J.,(1995) "*A parallel 2-opt algorithm for the Travelling Salesman Problem"* Published in "*Future Generation Computer Systems II"*, Elsevier B.V.

[96FM] Freisleben B., Merz P., (1996) *"A Genetic Local Search Algorithm For Solving Symmetric and Asymmetric Travelling Salesman Problems"* Published in Evolutionary Computation, 1996., Proceedings of IEEE International Conference

[SH20] Stütztle T., Hoos., H. H., (2000) *"Max-Min Ant System"* Elsevier Inc

[N03] Nilsson C., (2003) *"Heuristics for the Travelling Salesman Problem"* Linköping University

[JM95] Johnson D. S., McGeoch L. A., (1995) *"The Travelling Salesman Problem: A Case Study in Local Optimization"* chapter in *"Local Search in Combinatorial Optimization"* John Wiley & Sons, Inc. New York, NY, USA ©1997

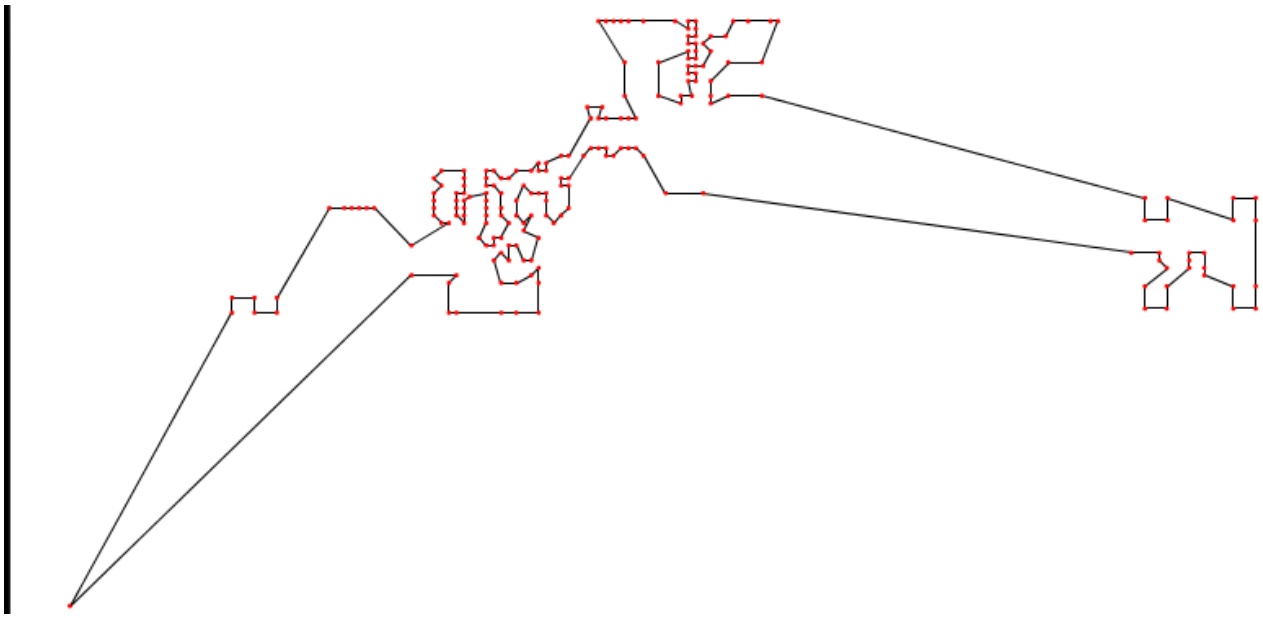
12: Appendix

In the appendix we have included pictures of the tested instances, step by step improvement of ant.tsp, our own code and the stdDraw.java and drawTour code we used for visualization of our instances by Keld Helsgaun. At the last page the cd with our code can be found.

Appendix A – Screen shots of instance solutions:

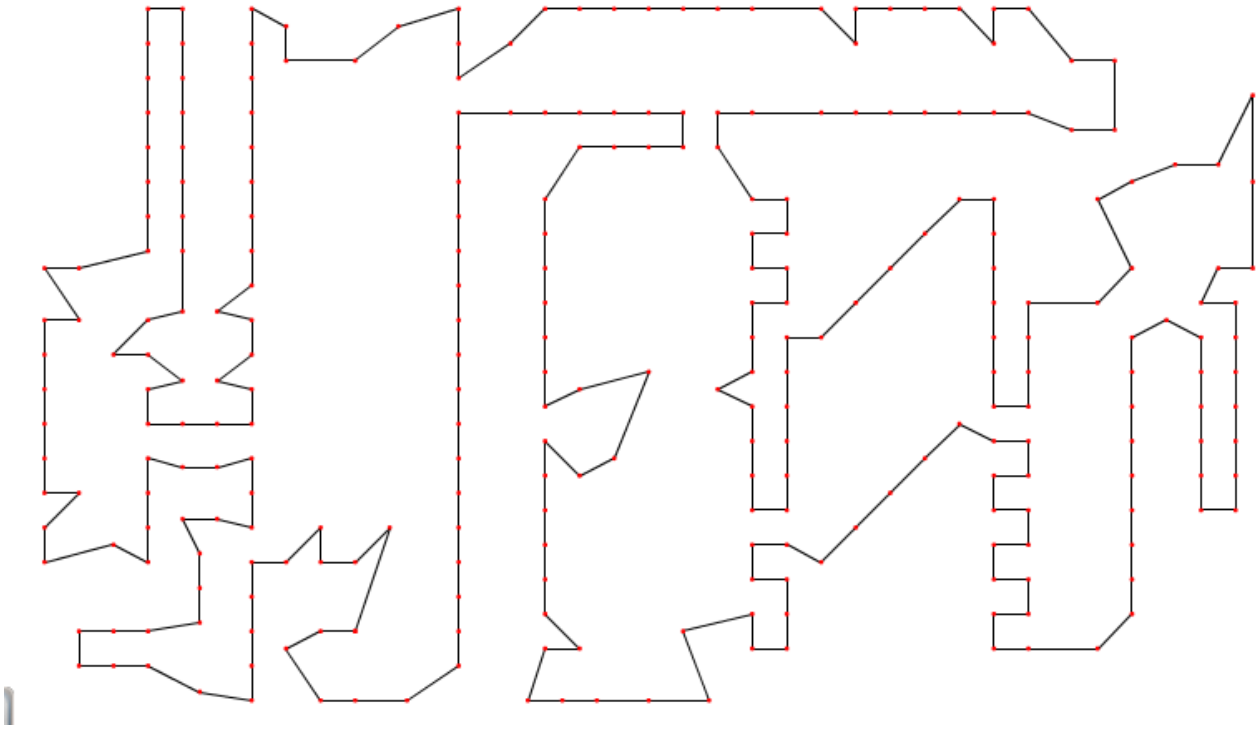
Here are some examples of what a user executing our application should expect to see visualized for different instances:

1. d198.tsp – optimum tour



2. a280.tsp

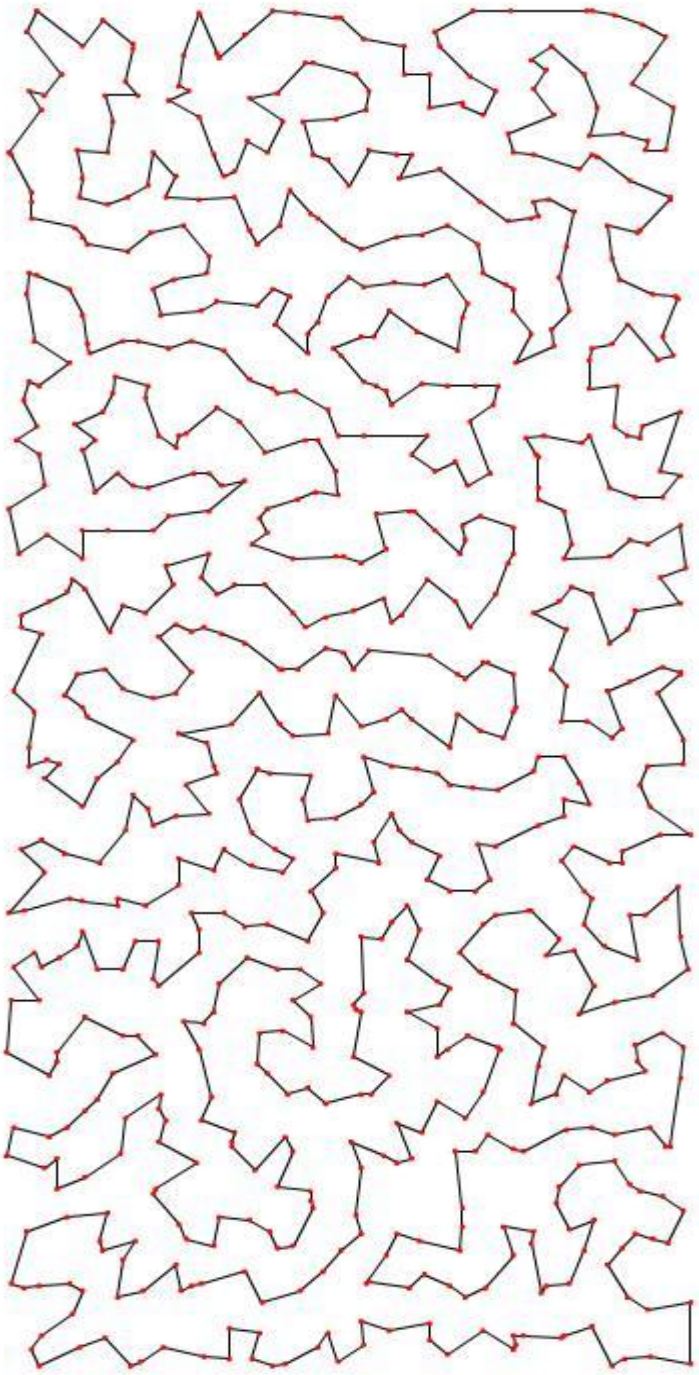
finding the optimum tour



```
aris@ariX:~/IdeaProjects/AsTSP/src$ java AsTSP -ia280.tsp  -m20 -e20 -l1 -o2579
-cm -t10000
Best tour length = 2757, after: 0 iterations,  gap = 6.902%
Best tour length = 2741, after: 2 iterations,  gap = 6.282%
Best tour length = 2739, after: 8 iterations,  gap = 6.204%
Best tour length = 2716, after: 15 iterations,  gap = 5.312%
Best tour length = 2690, after: 32 iterations,  gap = 4.304%
Best tour length = 2676, after: 35 iterations,  gap = 3.761%
Best tour length = 2665, after: 72 iterations,  gap = 3.335%
Best tour length = 2663, after: 84 iterations,  gap = 3.257%
Best tour length = 2635, after: 88 iterations,  gap = 2.171%
Best tour length = 2603, after: 144 iterations,  gap = 0.931%
Best tour length = 2601, after: 158 iterations,  gap = 0.853%
Best tour length = 2590, after: 198 iterations,  gap = 0.427%
Best tour length = 2585, after: 207 iterations,  gap = 0.233%
Best tour length = 2583, after: 225 iterations,  gap = 0.155%
Best tour length = 2579, after: 274 iterations,  gap = 0.000%
 :)     found optimal!!!   :)
run time: 18 sec
```

3. rat783.tsp using the MMAS algorithm with 2.5-opt

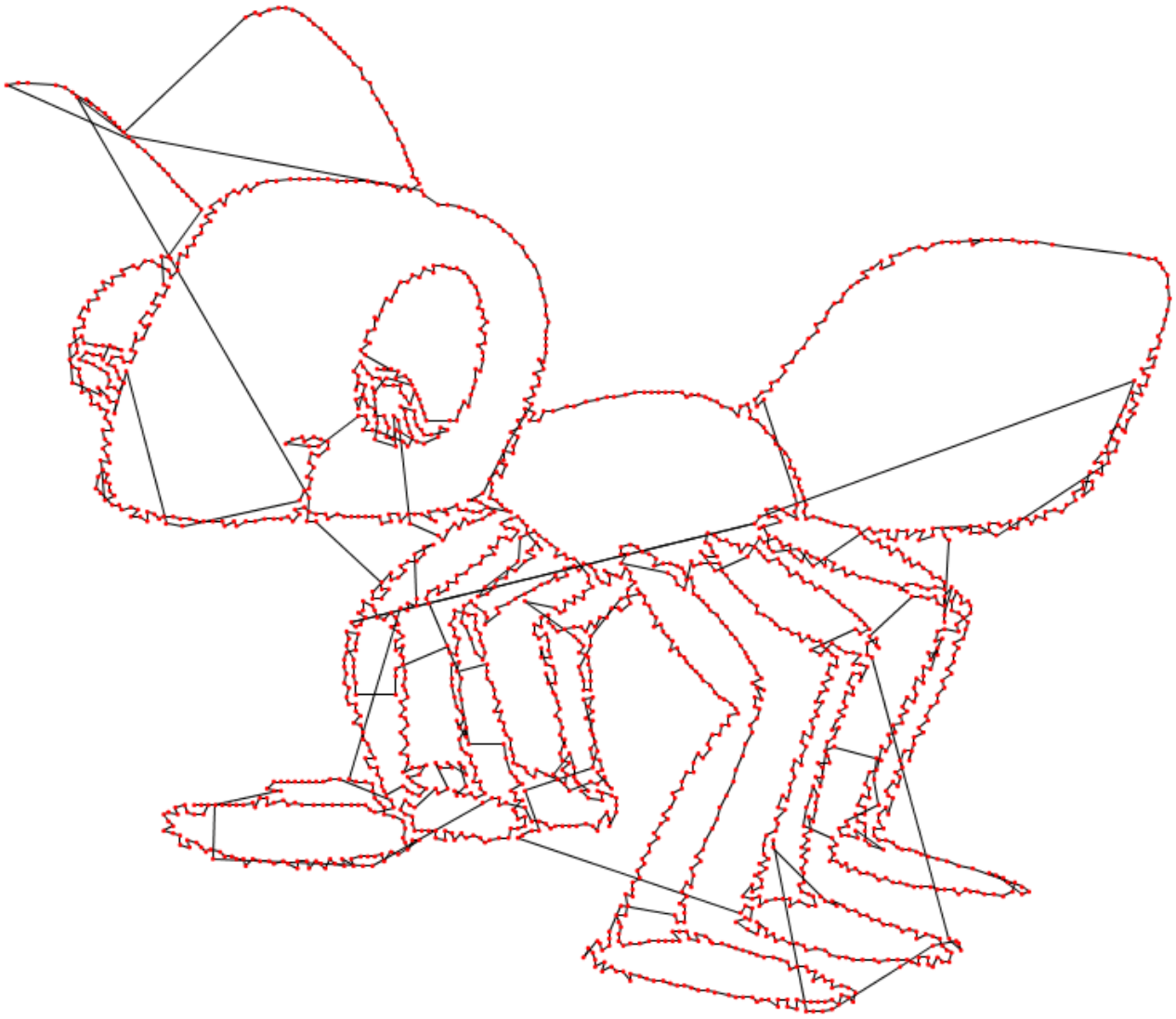
   The tour length is 8858 and the gap from the optimum tour is 0.591%



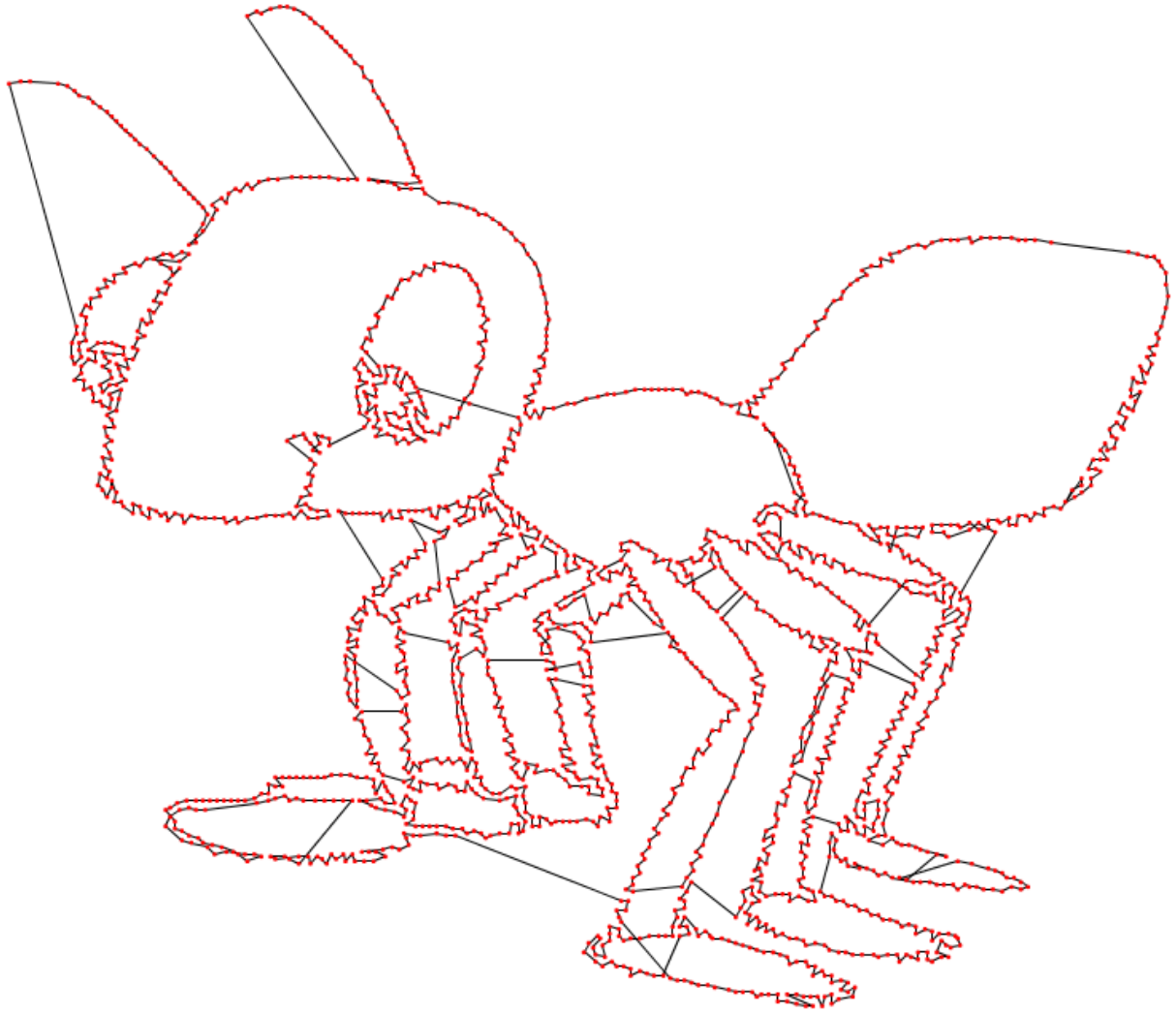
ant2038.tsp witch should create an image of an ant when the map of the optimum tour is designed (the instances are given by our supervisor Keld …)

As we can see the quality of the picture varies at the deferent gaps from the optimum:

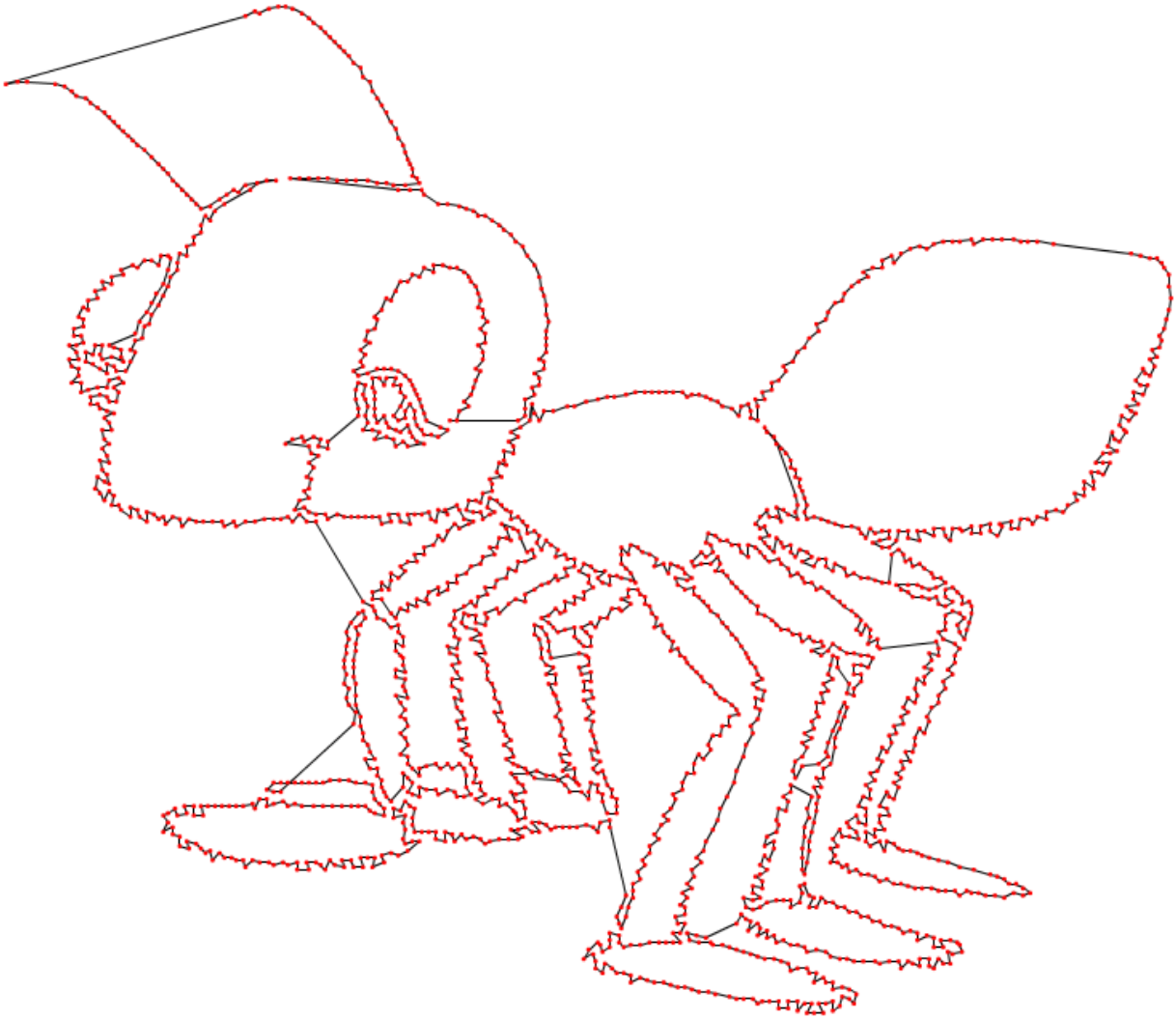
Here the tour length is 9921 and the gap from the optimum tour is 35.960%



Here the tour length is 8007 and the gap from the optimum tour is 12.348%



Here the tour length is and the gap from the optimum tour is %



Here the tour length 7467 is and the gap from the optimum tour is 2,33%

Appendix B. Our application Java Code

B.A: AsTSP class

```
1.  import java.util.ArrayList;
2.  import java.util.List;
3.  import java.util.Random;
4.  import java.io.File;
5.  import java.io.FileNotFoundException;
6.  import java.util.Scanner;
7.  import java.util.logging.Level;
8.  import static java.lang.System.exit;
9.  import static java.lang.System.nanoTime;
10. import static java.lang.System.out;
11. import static java.util.logging.Logger.getLogger;
12.
13. /**
14.  * Created by IntelliJ IDEA.
15.  * User: aris
16.  * Date: 4/17/11
17.  * Time: 8:48 PM
18.  * To change this template use File | Settings | File Templates.
19.  */
20.
21. public class AsTSP {
22.
23.     private int n;     //number of cities
24.     private int m;     //number of Ants
25.     private int nn;
26.     private int[][] distance;
27.     private int[][] nnList;
28.     private double[][] pheromone;
29.     private double[][] choice_info;
30.     private double[][] heuristicFactor;
31.     private int alpha;  //weight given to pheromone factor
32.     private int beta;   //weight given to heuristic factor
33.     private int e;      //weight given to the best tour
34.     private int local;
35.     private Ant[] ants;
36.     private Ant eliteAnt;
37.     private int bestTourLength = Integer.MAX_VALUE;
38.     private int LastbestTourLength = Integer.MAX_VALUE;
39.     private double[] x;  //coordinates
40.     private double[] y;
41.     private int[] bestTour;
42.     private int optimalTour;
43.     private double tmax;    //pheromone limits used in the MMas
44.     private double tmin;
```

```
45.    private char algorithm;
46.    private int nntl;  //Nearest Neighbor Tour Length
47.    private  double p;  //evaporation factor
48.    private  double q0;
49.    private Random rand=new Random();
50.
51.    /*
52.    Runs the main loop
53.    */
54.    public AsTSP(int m, String city_choice, int local, int a, int b, int o, int nn, int iterations,
     int e, char algorithm, double p, double q0) {
55.        this.m=m;
56.        this.local=local;
57.        this.alpha=a;
58.        this.beta=b;
59.        this.optimalTour=o;
60.        this.nn=nn;
61.        this.e=e;
62.        this.algorithm=algorithm;
63.        this.q0=q0;
64.        if (algorithm=='m')
65.           this.p=0.02;
66.        else
67.           this.p=p;
68.        InitializeData(city_choice);
69.
70.
71.        for (int i=0; i<iterations; i++)
72.        {
73.           List <Thread> threads = new ArrayList<Thread>();   //an arayList with all the
     threads
74.           for(int k=0; k<m; k++) {
75.              threads.add(k,new Thread(ants[k]));      //for every ant create a new thread and
     start it (construct tours)
76.              threads.get(k).start();
77.           }
78.           for (Thread t: threads)
79.           {
80.              try {
81.                 t.join();          //wait until all the treads are finished
82.              } catch (InterruptedException d) { d.printStackTrace(); }
83.           }
84.           UpdateStatistics(i);   //find the new best tour length
85.           if (bestTourLength<=optimalTour) {
86.              out.println(" :)     found optimal!!!   :)");
87.              break;          //if the optimal is found the program terminates
88.           }
89.           ASPheromoneUpdate();  //update the pheromones
```

```
90.             ComputeChoiceInformation();    //compute the choice info with the new pheromone
    values
91.         }
92.     }
93.
94.     /*
95.     Initialize all data to its initial value needed for the ants to construct the tour.
96.     */
97.     public void InitializeData(String fileName) {
98.
99.         ReadInstance(fileName);
100.                Ant[] ants = new Ant[m];
101.                double[][] choice_info = new double [n][n];
102.                double[][] pheromone = new double[n][n];
103.                int[][] distance = new int[n][n];
104.                int[] bestTour= new int [n+1];
105.                double[][] heuristicFactor = new double[n][n];
106.                int[][] nnList = new int [n][nn];
107.
108.                this.ants=ants;
109.                this.choice_info=choice_info;
110.                this.pheromone=pheromone;
111.                this.distance=distance;
112.                this.nnList=nnList;
113.                this.bestTour=bestTour;
114.                this.heuristicFactor=heuristicFactor;
115.
116.                ComputeDistances(x, y);
117.                ComputeNearestNeighborLists();
118.                nntl = ComputeNearestNeighborTourLength();
119.                InitializeAnts();
120.                InitializeStatistics();
121.                InitializePheromoneTrail();
122.                InitializeHeuristicFactor();
123.                ComputeChoiceInformation();
124.            }
125.
126.            /*
127.            Uses the scanner to read the instance file. What is read from the file is the x and y
    coordinates for the instance and the number of cities.
128.            */
129.            public void ReadInstance(String fileName)
130.            {
131.                File file = new File(fileName);
132.                try {
133.                    //
134.                    // Create a new Scanner object which will read the data
135.                    // from the file passed in. To check if there are more
```

```
136.                 // line to read from it we check by calling the
137.                 // scanner.hasNextLine() method. We then read line one
138.                 // by one till all line is read.
139.                 //
140.                 Scanner instanceScanner = new Scanner(file);
141.
142.                 n = instanceScanner.nextInt();
143.                 double[] x = new double[n];
144.                 double[] y = new double[n];
145.
146.                 this.x = x;
147.                 this.y = y;
148.
149.                 for (int i = 0; i < n; i++) {
150.                    int j = instanceScanner.nextInt() - 1;
151.
152.                    x[j] = instanceScanner.nextDouble();   //reading coordinates
153.                    y[j] = instanceScanner.nextDouble();
154.                 }
155.
156.              } catch (FileNotFoundException ex) {
157.              getLogger(AsTSP.class.getName()).log(Level.SEVERE, null, ex);  }
158.           }
159.
160.           /*
161.           Computes a matrix containing every distance to a city j when in a city i using the x
    and y coordinates
162.           */
163.           public void ComputeDistances(double[] x, double[] y)
164.           {
165.              for ( int i=0; i<n; i++)
166.                 for ( int j=i+1; j<n; j++) {
167.                       double dx = x[i] - x[j], dy = y[i] - y[j];
168.                       distance[i][j] = (int) (Math.sqrt(dx * dx + dy * dy)+0.5);  //round the
    distance to the closest int
169.                       distance[j][i]=distance[i][j];
170.                 }
171.           }
172.
173.           /*
174.           Computes the nnList which contains the nn nearest city for each city.
175.           */
176.           public void ComputeNearestNeighborLists()
177.           {
178.
179.              int[][] sorted_dis = new int[n][nn];  //an array with the nn smaller distances
180.              boolean[] sorted = new boolean[n];    //saws if the city i is already at the
    sorted_dis
```

```
181.            boolean  swapped;
182.
183.            for (int i=0; i<n; i++)
184.               for(int j=0; j<nn; j++) {
185.                  if (i!=j) {
186.                     nnList[i][j]=j;       //  nnList is the index that saws which city each
    distance represents
187.                     sorted_dis[i][j]=distance[i][j];
188.                     sorted[j]=true;
189.                  }
190.                  else if(i==j && (j+nn)<n){
191.                     nnList[i][j]=j+nn;
192.                     sorted_dis[i][j]=distance[i][j+nn];
193.                     sorted[j+nn]=true;
194.                  }
195.                  else if(i==j && (j+nn)>n){
196.                     nnList[i][j]=nn-j;
197.                     sorted_dis[i][j]=distance[i][nn-j];
198.                     sorted[j-nn]=true;
199.                  }
200.               }
201.
202.            for (int i=0; i<n; i++) {       //sort the sorted_dis array
203.               do {
204.                  swapped = false;
205.                  for(int j=1; j<nn; j++) {
206.                     if (sorted_dis[i][j-1]>sorted_dis[i][j]) {
207.                        int temp = sorted_dis[i][j];
208.                        int temp2 = nnList[i][j];
209.                        sorted_dis[i][j] = sorted_dis[i][j-1];
210.                        nnList[i][j] = nnList[i][j-1];
211.                        sorted_dis[i][j-1] = temp;
212.                        nnList[i][j-1] = temp2;
213.                        swapped = true;
214.                     }
215.                  }
216.               } while (swapped);
217.            }
218.
219.            for (int i=0; i<n; i++) {
220.               for (int j=0; j<n; j++) {
221.                  sorted[j]=false;
222.               }
223.               for (int j=0; j<nn; j++) {
224.                  sorted[nnList[i][j]]=true;
225.               }
226.               sorted[i]=true;
227.
```

```
228.                for(int j=0; j<n; j++) {           //for every city
229.                   if (distance[i][j] < sorted_dis[i][nn-1] && sorted[j]==false) {  //it checks if
    the distance is smaller and this city is not yet sorted
230.                      sorted[nnList[i][nn-1]]=false;
231.                      sorted_dis[i][nn-1]=distance[i][j];
232.                      nnList[i][nn-1]=j;
233.                      sorted[j]=true;
234.                      for (int s=0; s<nn-1; s++) {    //sort function
235.                         if(sorted_dis[i][s]>sorted_dis[i][nn-1]) {   //add the city to the
    sorted_did array to the rigth place
236.                            int temp=sorted_dis[i][nn-1];          //and delete the largest distance
237.                            int temp2=nnList[i][nn-1];
238.                            for (int d=nn-1; d>s; d--) {
239.                               sorted_dis[i][d]=sorted_dis[i][d-1];
240.                               nnList[i][d]=nnList[i][d-1];
241.                            }
242.                            sorted_dis[i][s]=temp;
243.                            nnList[i][s]=temp2;
244.                            break;
245.                         }
246.                      }
247.                   }
248.                }
249.             }
250.          }
251.
252.          /*
253.          Finds the length of a tour found by using the nearest neighbor algorithm used to
    compute some other data.
254.          */
255.          public int ComputeNearestNeighborTourLength()
256.          {
257.             int prev_city=0, next_city, minDis;
258.             int minCit=0;
259.             int tourLength=0;
260.             boolean[] visited = new boolean[n];
261.
262.             visited[prev_city] = true;
263.
264.             for (int step=1; step < n; step++)
265.             {
266.                minDis=Integer.MAX_VALUE;
267.                for (int j=0; j<n; j++) {
268.                   next_city = j;
269.                   if ( prev_city==next_city )
270.                      continue;
271.                   else if (visited[next_city] == false && distance[prev_city][next_city] <
    minDis) {
```

```
                    minDis = distance[prev_city][next_city];
                    minCit=next_city;
                }
            }
            tourLength += distance[prev_city][minCit];
            prev_city=minCit;
            visited[prev_city]=true;
        }
        tourLength += distance[prev_city][0];

        return tourLength;
    }

    /*
    Initialize the amount of pheromone on all edges to 1 divided by the nearest neighbor tourlength.
    */
    public void InitializePheromoneTrail()
    {
        if (algorithm=='a')
            for (int i=0; i<n; i++)
                for(int j=i+1; j<n; j++) {
                    pheromone[i][j] = (double)m / (double)nntl;
                    pheromone[j][i] = pheromone[i][j];
                }
        else if (algorithm=='e')
            for (int i=0; i<n; i++)
                for(int j=i+1; j<n; j++) {
                    this.pheromone[i][j] = (double)(e+m)/(p * (double)nntl);
                    pheromone[j][i] = pheromone[i][j];
                }
        else if (algorithm=='m')
            for (int i=0; i<n; i++)
                for(int j=i+1; j<n; j++) {
                    this.pheromone[i][j] = tmax;
                    pheromone[j][i] = pheromone[i][j];
                }
    }

    /*
    Computes a heuristicFactor matrix containing all heuristic information for all edges.
    */
    public void InitializeHeuristicFactor()
    {
        for (int i=0; i<n; i++)
            for(int j=i+1; j<n; j++) {
                heuristicFactor[i][j] =  1.0 / (distance[i][j]+0.000001);  //divide with a non zero value
```

```
318.                    heuristicFactor[j][i]=heuristicFactor[i][j];
319.                }
320.            }
321.
322.            /*
323.            Computes the ChoiceInformation for every edge using the pheromone matrix and
    the heuristicFactor matrix to calculate the choice info.
324.            */
325.            public void ComputeChoiceInformation()
326.            {
327.                for (int i=0; i<n; i++)
328.                    for(int j=i+1; j<n; j++) {
329.                        choice_info[i][j] = Math.pow(pheromone[i][j], alpha) *
    Math.pow(heuristicFactor[i][j], beta);
330.                        choice_info[j][i] = choice_info[i][j];
331.                    }
332.            }
333.
334.            /*
335.            Creates a number of new ant objects equal to the size of m. Each ant is given the
    number of cities, the distance matrix and the choice info matrix.
336.             */
337.            public void InitializeAnts()
338.            {
339.                for (int i=0; i<m; i++)
340.                    ants[i] = new Ant(n, distance, choice_info, nnList, nn, local);
341.            }
342.
343.            /*
344.            Creates an eliteAnt object containing the same information as a normal ant.
345.            */
346.            public void InitializeStatistics()
347.            {
348.                Ant eliteAnt = new Ant(n, distance, choice_info, nnList, nn, local);
349.                this.eliteAnt=eliteAnt;
350.                tmax=1.0/(p*nntl);
351.            }
352.
353.            /*
354.            Checks whether each ant have found a better tourLength than the bestTourLength.
    If so the bestTourLength is set equal to the new best tourLengths value. Also the elite ant is
    set equal to the ant who found the best tour, BestTourLength, current iteration and deviation
    from the gap is printed.
355.            */
356.            public void UpdateStatistics(int i)
357.            {
358.                int IterationCount=0;
359.                for (int k=0; k<m; k++) {
```

```
360.                ants[k].cal_tour_length();
361.                if (LastbestTourLength>ants[k].getTourLength()) {
362.                   eliteAnt=ants[k];
363.                   LastbestTourLength=eliteAnt.getTourLength();
364.                }
365.             }
366.             if (bestTourLength>LastbestTourLength) {
367.                bestTourLength=LastbestTourLength;
368.                if (algorithm=='m') {
369.                   for (int j=0; j<n+1; j++)
370.                      bestTour[j]=eliteAnt.getTourCity(j);
371.                   tmax=1.0/(p*bestTourLength);
372.                   tmin= tmax*( Math.pow(0.05, 1.0/n))/(((n/2)-1)*Math.pow(0.05, 1.0/n));
373.                   IterationCount=0;
374.                }
375.                out.printf("Best tour length = %d, after: %d iterations,  gap = %2.3f%%\n",
    bestTourLength, i, ((double)(bestTourLength-optimalTour)*100.0)/(double)optimalTour);
376.                eliteAnt.draw(x,y,0);  //print the map of the best so far tour
377.             }
378.             else if (algorithm=='m') {
379.
380.                IterationCount++;
381.
382.                if (IterationCount>60) {         //if after 60 iteration not a better tour been found
    re-initialize the pheromone matrix
383.                   IterationCount=0;
384.                   InitializePheromoneTrail();
385.                }
386.             }
387.             LastbestTourLength=Integer.MAX_VALUE;
388.          }
389.
390.          /*
391.          Calls the evaporate and the deposit Pheromone.
392.          */
393.          public void ASPheromoneUpdate()
394.          {
395.             if (algorithm=='a' || algorithm=='e') {
396.             Evaporate();
397.             for (int k=0; k<m; k++)
398.                DepositPheromone(k);
399.             }
400.             if (algorithm=='e')
401.                EliteDeposit();   //the elite ants deposits extra pheromone toString() its args
402.             if (algorithm=='m') {
403.                MMEvaporate();
404.                MMDeposit();
405.             }
```

```
406.        }
407.
408.        /*
409.        Evaporates some of the pheromone on every edge at each iteration.
410.        */
411.        public void Evaporate()
412.        {
413.           for (int i=0; i<n; i++)
414.              for (int j=i+1; j<n; j++) {
415.                 pheromone[i][j] = (1.0-p) * pheromone[i][j];
416.                 pheromone[j][i] = pheromone[i][j];
417.              }
418.        }
419.
420.        /*
421.        Deposit pheromone on the tour edges an ant an ant has been throw.
422.        */
423.        public void DepositPheromone(int k)
424.        {
425.           int j,l;
426.           double trail = 1.0/(double)ants[k].getTourLength();
427.
428.           for (int i=0; i<n; i++) {
429.              j = ants[k].getTourCity(i);
430.              l = ants[k].getTourCity(i+1);
431.              pheromone[j][l] = pheromone[j][l]+trail;
432.              pheromone[l][j] = pheromone[j][l];
433.           }
434.        }
435.
436.        /*
437.        Deposit extra pheromone on the best so far tours arcs. It deposit pheromone a
    number of times equal to a variable e which stands for eliteAnts.
438.        */
439.        public void EliteDeposit()
440.        {
441.           int i, j, l;
442.           double trail = (double)e/(double)eliteAnt.getTourLength();
443.           for (i=0; i<n; i++) {
444.              j = eliteAnt.getTourCity(i);
445.              l = eliteAnt.getTourCity(i+1);
446.              pheromone[j][l] = pheromone[j][l]+trail;
447.              pheromone[l][j] = pheromone[j][l];
448.           }
449.        }
450.
451.        public void MMDeposit()
452.        {
```

```
453.            int i, j, l;
454.            double q;           // here a pseudorandom rule is implemented
455.            q=rand.nextDouble();  //produce a random number between 0 an 1
456.            if (q0<q) {       //if that number is smaller than the parameter q0 the best-
    iteration-ant deposit pheromone
457.               double trail = 1.0/(double)eliteAnt.getTourLength();
458.               for (i=0; i<n; i++) {
459.                  j = eliteAnt.getTourCity(i);
460.                  l = eliteAnt.getTourCity(i+1);
461.                  pheromone[j][l] = pheromone[j][l]+trail;
462.                  if (pheromone[j][l]>tmax)      //the pheromones cant go higher than tmax
463.                     pheromone[j][l]=tmax;
464.                  pheromone[l][j] = pheromone[j][l];
465.               }
466.            }
467.            else {                  //the best-so-far ant deposit pheromone
468.               double trail = 1.0/(double)bestTourLength;
469.               for (i=0; i<n; i++) {
470.                  j = bestTour[i];
471.                  l = bestTour[i+1];
472.                  pheromone[j][l] = pheromone[j][l]+trail;
473.                  if (pheromone[j][l]>tmax)
474.                     pheromone[j][l]=tmax;
475.                  pheromone[l][j] = pheromone[j][l];
476.               }
477.            }
478.         }
479.
480.         public void MMEvaporate()
481.         {
482.            for (int i=0; i<n; i++)
483.               for (int j=i+1; j<n; j++) {
484.                  pheromone[i][j] = (1.0-p) * pheromone[i][j];
485.
486.                  if (pheromone[i][j]<tmin)   //the pheromones cant go higher than tmin
487.                     pheromone[i][j]=tmin;
488.                  pheromone[j][i] = pheromone[i][j];
489.               }
490.         }
491.
492.         /*
493.         The main class, also parse the parameters and calculates runtime.
494.         */
495.         public static void main(String[] args)
496.         {
497.            final long startTime = nanoTime();   //calculating run time function
498.            if(args.length==0) {
499.               out.println("No parameters!  -h for help");
```

```
500.              terminate();
501.           }
502.           parse_args(args);     //import the parameters
503.           final long duration = nanoTime() - startTime;  //  run time
504.              out.println("run time: "+duration/1000000000+" sec");
505.           terminate();
506.        }
507.
508.        /*
509.        Initialize all the parameters by searching for '-'symbol
510.         */
511.        public static void parse_args(String[] args)
512.        {              //set the defaultt parameters
513.           //char[] city =  {'d','1','9','8','.','t','s','p'};
514.           String city_choice=new String();
515.           char algorithm='e';
516.           char[] param;
517.           String parameter;
518.           int m=50, local=1, a=1, b=5, nn=20, o=0, iterations=1000, e=m;
519.           double p=0.5;   //for the MMAS algorithm p=0.02 is suggested
520.           double q0=0.9;
521.
522.           for (int i=0; i<args.length; i++) {
523.              if (args[i].contains("-i")) {
524.                 param = new char[args[i].length()-2];
525.                 args[i].getChars(2, args[i].length(), param, 0);
526.                 city_choice = new String(param);
527.              }
528.              else if (args[i].contains("-m")) {
529.                 param = new char[args[i].length()-2];
530.                 args[i].getChars(2, args[i].length(), param, 0);
531.                 parameter=new String(param);
532.                 m=Integer.parseInt(parameter);
533.              }
534.              else if (args[i].contains("-l")) {
535.                 param = new char[args[i].length()-2];
536.                 args[i].getChars(2, args[i].length(), param, 0);
537.                 parameter=new String(param);
538.                 local=Integer.parseInt(parameter);
539.              }
540.              else if (args[i].contains("-a"))  {
541.                 param = new char[args[i].length()-2];
542.                 args[i].getChars(2, args[i].length(), param, 0);
543.                 parameter=new String(param);
544.                 a=Integer.parseInt(parameter);
545.              }
546.              else if (args[i].contains("-b"))  {
547.                 param = new char[args[i].length()-2];
```

```
548.                 args[i].getChars(2, args[i].length(), param, 0);
549.                 parameter=new String(param);
550.                 b=Integer.parseInt(parameter);
551.              }
552.              else if (args[i].contains("-o"))  {
553.                 param = new char[args[i].length()-2];
554.                 args[i].getChars(2, args[i].length(), param, 0);
555.                 parameter=new String(param);
556.                 o=Integer.parseInt(parameter);
557.              }
558.              else if (args[i].contains("-nn"))  {
559.                 param = new char[args[i].length()-3];
560.                 args[i].getChars(3, args[i].length(), param, 0);
561.                 parameter=new String(param);
562.                 nn=Integer.parseInt(parameter);
563.              }
564.              else if (args[i].contains("-t"))  {
565.                 param = new char[args[i].length()-2];
566.                 args[i].getChars(2, args[i].length(), param, 0);
567.                 parameter=new String(param);
568.                 iterations=Integer.parseInt(parameter);
569.              }
570.              else if (args[i].contains("-e"))  {
571.                 param = new char[args[i].length()-2];
572.                 args[i].getChars(2, args[i].length(), param, 0);
573.                 parameter=new String(param);
574.                 e=Integer.parseInt(parameter);
575.              }
576.              else if (args[i].contains("-c"))
577.                 algorithm=args[i].charAt(2);
578.
579.              else if (args[i].contains("-p"))  {
580.                 param = new char[args[i].length()-2];
581.                 args[i].getChars(2, args[i].length(), param, 0);
582.                 parameter=new String(param);
583.                 p=Double.parseDouble(parameter);
584.              }
585.              else if (args[i].contains("-q"))  {
586.                 param = new char[args[i].length()-2];
587.                 args[i].getChars(2, args[i].length(), param, 0);
588.                 parameter=new String(param);
589.                 q0=Double.parseDouble(parameter);
590.              }
591.              else if (args[i].contains("-h")) {
592.                 out.println("Parameters list:\n-i&file name\n-m&numberof ants\n-l& 0 or 1
    to activate local surch\n-a&pheromone factor weight\n-b&(1-5) heuristic factor weight\n-
    o&otimum tour length\n-nn&number of nearest neighbors\n-t&number of iterations\n-
```

```
    p&evaporation factor value\n-e&elite and wheight\n-a&choose algorithm (a-simpleAS, e-
    EAS, m-MMAS)\n-q0 for pseudorandom factor");
593.                    terminate();
594.                }
595.            }
596.            if (city_choice.length()==0)
597.                city_choice=new String("d198.tsp");
598.            AsTSP demo = new AsTSP(m,city_choice, local, a, b, o, nn, iterations, e,
    algorithm, p, q0);
599.        }
600.
601.        public static void terminate()
602.        {
603.            exit(0);
604.        }
605.    }
```

Appendix B.B - ANT class

```
1.  import java.util.Random;
2.  /**
3.   * Created by IntelliJ IDEA.
4.   * User: aris
5.   * Date: 4/18/11
6.   * Time: 3:21 PM
7.   * To change this template use File | Settings | File Templates.
8.   */
9.
10. public class Ant implements Runnable{
11.
12.     private int n;
13.     private int tour_length;
14.     private int[] tour;
15.     private int[] index;
16.     private boolean[] visited = new boolean [n];
17.     private int[][] distance;
18.     private double[][] choice_info;
19.     private int[][] nnList;
20.     private int nn;
21.     private int local;
22.     private Random rand=new Random();
23.
24.     /*
25.     The constructor of the ant objects.
26.     */
27.     public Ant(int n, int[][] distance, double[][] choice_info, int[][] nnList, int nn, int local) {
```

```
28.
29.     int[] tour_index = new int [n];
30.     boolean[] visited = new boolean [n];
31.     int[] tour = new int [n+1];
32.
33.     this.n = n;
34.     this.nnList=nnList;
35.     this.nn=nn;
36.     this.local=local;
37.     this.distance = distance;
38.     this.choice_info = choice_info;
39.     this.index=tour_index;
40.     this.tour=tour;
41.     this.visited=visited;
42.
43.     for(int i=0; i<n; i++)
44.         this.index[i]=i;
45.  }
46.
47.  /*
48.  Calculates the tour length when called.
49.  */
50.  public void cal_tour_length()
51.  {
52.     this.tour_length = 0;
53.     for(int i=0; i < n; i++)
54.         tour_length += distance[tour[i]][tour[i+1]];
55.  }
56.
57.  /*
58.  Getter for the tourLength.
59.  */
60.  public final int getTourLength()
61.  {
62.     return tour_length;
63.  }
64.
65.  public final int[] getTour()
66.  {
67.     return tour;
68.  }
69.
70.  /*
71.  Getter for specific tour[i]
72.  */
73.  public final int getTourCity(int i)
74.  {
75.     return tour[i];
```

```
76.     }
77.
78.         /*
79.     Construct a new tour for each ant. Also reset the visited array for every ant and place
        them on a new random city.
80.     */
81.     public void ConstructSolutions()
82.     {
83.         for (int i=0; i<n; i++)
84.             visited[i]=false;
85.
86.         int r = (int) (rand.nextDouble() * n);
87.         visited[r]=true;
88.         tour[0]=r;
89.         index[r]=0;
90.
91.         for (int i=1; i<n; i++) {
92.             //ants[k].ASDecisionRule(i);
93.             NeighborListASDecisionRule(i);
94.         }
95.         tour[n]=tour[0];
96.         if (local==1) {
97.             Opt2();
98.             Opt2_5();
99.
100.                }
101.            }
102.
103.            /*
104.            Prints out the map of the tour.
105.            */
106.            public void draw(double [] x, double[] y, int delay)
107.            {
108.                DrawTour draw = new DrawTour();
109.                draw.draw(tour,x,y,0);
110.            }
111.            /*
112.            Decides which city an ant in a current city should go to next. Uses the nearest
        neighbor list to speed up the decision.
113.            */
114.            public void NeighborListASDecisionRule(int step)
115.            {
116.                int c, j;
117.
118.                double sum_probabilities;
119.                double[] selection_probability= new double[n];
120.                double r, p;
121.
```

```
122.            c = tour[step-1];
123.            sum_probabilities=0.0;
124.
125.            for (j=0; j<nn; j++) {
126.               if ( visited[nnList[c][j]] == true )
127.                  selection_probability[j]=0.0;
128.               else {
129.                  selection_probability[j]=choice_info[c][nnList[c][j]];
130.                  sum_probabilities = sum_probabilities + selection_probability[j];
131.               }
132.            }
133.            if (sum_probabilities==0)
134.               ChooseBestNext(step);
135.            else {
136.               r = rand.nextDouble() *  sum_probabilities;
137.               j=0;
138.               p = selection_probability[j];
139.               while (p<r) {
140.                  j++;
141.                  p = p+selection_probability[j];
142.               }
143.               tour[step] = nnList[c][j];
144.               index[nnList[c][j]]=step;
145.               visited[nnList[c][j]] =  true;
146.            }
147.         }
148.
149.         /*
150.         Used by the NeighborListASDecisionRule to make the ant go to the city with the
     highest choice information when in a current city.
151.         */
152.         public void ChooseBestNext(int step)
153.         {
154.            int c, j, nc=0;
155.            double u=0.0;
156.
157.            c = tour[step-1];
158.            for(j=0; j<n; j++)
159.               if ( visited[j] == false )
160.                  if (choice_info[c][j] > u ) {
161.                     nc = j;
162.                     u = choice_info[c][j];
163.                  }
164.            tour[step] = nc;
165.            index[nc]=step;
166.            visited[nc] = true;
167.         }
168.
```

```
169.        /*
170.        Performs localSearch to the found tour by the ant.
171.        */
172.        public void Opt2_5()
173.        {
174.           double maxGain;
175.           double gain;
176.           boolean isChange;
177.           boolean[] dontLookBit = new boolean[n];
178.
179.           for (int j=0; j<n; j++)
180.              dontLookBit[j]=false;
181.
182.           do {
183.           isChange=false;
184.
185.           for (int i=1; i<n-1; i++) {  //for every city try 2-opt
186.              if (dontLookBit[i-1]==false) {
187.                 int a = tour[i-1];
188.                 int b = tour[i];
189.                 int e = tour[i+1];
190.                 int c,d;
191.                 maxGain = 0;
192.                 gain = 0;
193.                 int dPos=0;
194.                 for (int j=i+2; j<n-1; j++) {
195.                    if ( (j!=i) && (j != (i-1)) && (j != (i+1)) && ((j-1)!=i) && ((j-1) != (i-1))
    && ((j-1) != (i+1)) )
196.                    {
197.                       c = tour[j];
198.                       d = tour[j-1];
199.                       //  calculate if there is gain
200.
201.                       gain = ( distance[a][b] + distance[b][e] + distance[d][c] ) - (
    distance[a][e] + distance[d][b] + distance[b][c] );
202.
203.                       if (gain > maxGain) {    // find the most gainful move
204.                          maxGain = gain;
205.                          dPos = j-1;
206.                       }
207.                    } //end if
208.                 } //end for j
209.                 if (maxGain > 0 ) {
210.                    if (i < dPos) {
211.                       move(i, dPos);
212.                    }
213.                    isChange = true;
214.                    dontLookBit[i-1]=false;
```

```
215.                    dontLookBit[i]=false;
216.                    dontLookBit[i+1]=false;
217.                    dontLookBit[dPos]=false;
218.                    dontLookBit[dPos+1]=false;
219.
220.                } //end if
221.                else {
222.                    dontLookBit[i-1]=true;
223.                }
224.             }
225.          }// end for i
226.       }while (isChange==true);
227.          //System.out.println("*******out of do while**********");
228.       }
229.
230.       public void move (int bPos, int dPos) {
231.          int temp=tour[bPos];
232.          for(int i=bPos; i<dPos; i++) {
233.             tour[i]=tour[i+1];
234.          }
235.          tour[dPos]=temp;
236.
237.       }
238.
239.       /*
240.       Performs localSearch to the found tour by the ant.
241.       */
242.       public void Opt2()
243.       {
244.          double maxGain;
245.          double gain;
246.          boolean isChange;
247.          boolean[] dontLookBit = new boolean[n+1];
248.
249.          for (int j=0; j<=n; j++)
250.             dontLookBit[j]=false;
251.
252.          do {
253.          isChange=false;
254.
255.          for (int i=1; i<n; i++) {  //for every city try 2-opt
256.             if (dontLookBit[i-1]==false) {
257.                int a = tour[i-1];
258.                int b = tour[i];
259.                int c,d;
260.                maxGain = 0;
261.                gain = 0;
262.                int maxEdge = 0;
```

```
263.                    for (int j=1; j<=n; j++) {
264.                       if ( (j!=i) && ((j-1)!=i) && ((j-1) != (i-1)) && (j != (i-1)))
265.                       {
266.                          c = tour[j];
267.                          d = tour[j-1];
268.                          //  calculate if there is gain
269.
270.                          gain = ( distance[a][b] + distance[c][d] ) - ( distance[a][d] +
    distance[b][c] );
271.
272.                          if (gain > maxGain) {    // find the most gainful move
273.                             maxGain = gain;
274.                             maxEdge = j-1;
275.                          }
276.                       } //end if
277.                    } //end for j
278.                    if (maxGain > 0 ) {     // if there is gain make the 2-opt move (flip 2 cities)
279.                       int u;
280.                       int v;
281.
282.                       for ( u = maxEdge, v = i; u>=i;  u--, v++)
283.                       {
284.                          if (v < u) {        //  change also the order of the cities (u--)
285.                             flip(v, u);
286.                          }
287.                       }
288.                       isChange = true;
289.                       dontLookBit[i-1]=false;
290.                       dontLookBit[i]=false;
291.                       dontLookBit[maxEdge]=false;
292.                       dontLookBit[maxEdge+1]=false;
293.                    } //end if
294.                    else {
295.                       dontLookBit[i-1]=true;
296.                    }
297.                 }
298.              }// end for i
299.           }while (isChange==true);
300.           }
301.
302.           // flip 2 cities in the tour array. Used by the 2-opt method.
303.           public void flip (int i, int j)
304.           {
305.              int temp1, temp2;
306.              int a=tour[i], b=tour[j];
307.
308.              temp1 = tour[i];
309.              temp2 = index[a];
```

```
310.
311.                tour[i] = tour[j];
312.                index[a] = index[b];
313.
314.                tour[j] = temp1;
315.                index[b] = temp2;
316.            }
317.
318.            public void run() {
319.                ConstructSolutions();
320.            }
321.        }
```

Appendix C - Java classes created from our supervisor Keld Helsgaun that we use in order to print the tour map

Appendix C.A - StdDraw

```
1.  import java.io.*;
2.  import java.net.*;
3.  import java.awt.*;
4.  import java.awt.geom.*;
5.  import java.awt.event.*;
6.  import java.awt.image.*;
7.  import javax.swing.*;
8.  import javax.imageio.ImageIO;
9.  import java.util.LinkedList;
10. import javax.imageio.ImageIO;
11. import java.util.LinkedList;
12.
13. /**
14.  * Created by IntelliJ IDEA.
15.  * User: Asger
16.  * Date: 11-05-11
17.  * Time: 18:58
18.  * To change this template use File | Settings | File Templates.
19.  */
20.
21.
22.
23. /*************************************************************************
24.  *  Compilation:  javac StdDraw.java
25.  *  Execution:    java StdDraw
```

```
 *
 *  Standard drawing library. This class provides a basic capability for
 *  creating drawings with your programs. It uses a simple graphics model that
 *  allows you to create drawings consisting of points, lines, and curves
 *  in a window on your computer and to save the drawings to a file.
 *
 *  Todo
 *  ----
 *    - Add support for gradient fill, etc.
 *
 *  Remarks
 *  -------
 *    - don't use AffineTransform for rescaling since it inverts
 *      images and strings
 *    - careful using setFont in inner loop within an animation -
 *      it can cause flicker
 *
 ************************************************************************/


/**
 *  <i>Standard draw</i>. This class provides a basic capability for
 *  creating drawings with your programs. It uses a simple graphics model that
 *  allows you to create drawings consisting of points, lines, and curves
 *  in a window on your computer and to save the drawings to a file.
 *  <p>
 *  For additional documentation, see <a
href="http://www.cs.princeton.edu/introcs/15inout">Section 1.5</a> of
 *  <i>Introduction to Programming in Java: An Interdisciplinary Approach</i> by Robert
Sedgewick and Kevin Wayne.
 */
public final class StdDraw implements ActionListener, MouseListener,
MouseMotionListener, KeyListener {

   // pre-defined colors
   public static final Color BLACK      = Color.BLACK;
   public static final Color BLUE       = Color.BLUE;
   public static final Color CYAN       = Color.CYAN;
   public static final Color DARK_GRAY  = Color.DARK_GRAY;
   public static final Color GRAY       = Color.GRAY;
   public static final Color GREEN      = Color.GREEN;
   public static final Color LIGHT_GRAY = Color.LIGHT_GRAY;
   public static final Color MAGENTA    = Color.MAGENTA;
   public static final Color ORANGE     = Color.ORANGE;
   public static final Color PINK       = Color.PINK;
   public static final Color RED        = Color.RED;
   public static final Color WHITE      = Color.WHITE;
   public static final Color YELLOW     = Color.YELLOW;
```

```
71.
72.    /**
73.     * Shade of blue used in Introduction to Programming in Java.
74.     * It is Pantone Pantone 300U. The RGB values are approximately (9, 90, 266).
75.     */
76.    public static final Color BOOK_BLUE       = new Color(  9,  90, 166);
77.    public static final Color BOOK_LIGHT_BLUE = new Color(103, 198, 243);
78.
79.    /**
80.     * Shade of red used in Algorithms 4th edition.
81.     * It is Pantone 1805U. The approximate RGB values are (150, 35, 31).
82.     */
83.    public static final Color BOOK_RED = new Color(150, 35, 31);
84.
85.    // default colors
86.    private static final Color DEFAULT_PEN_COLOR   = BLACK;
87.    private static final Color DEFAULT_CLEAR_COLOR = WHITE;
88.
89.    // current pen color
90.    private static Color penColor;
91.
92.    // default canvas size is SIZE-by-SIZE
93.    private static final int DEFAULT_SIZE = 512;
94.    private static int width  = DEFAULT_SIZE;
95.    private static int height = DEFAULT_SIZE;
96.
97.    // default pen radius
98.    private static final double DEFAULT_PEN_RADIUS = 0.002;
99.
100.           // current pen radius
101.           private static double penRadius;
102.
103.           // show we draw immediately or wait until next show?
104.           private static boolean defer = false;
105.
106.           // boundary of drawing canvas, 5% border
107.           private static final double BORDER = 0.05;
108.           private static final double DEFAULT_XMIN = 0.0;
109.           private static final double DEFAULT_XMAX = 1.0;
110.           private static final double DEFAULT_YMIN = 0.0;
111.           private static final double DEFAULT_YMAX = 1.0;
112.           private static double xmin, ymin, xmax, ymax;
113.
114.           // for synchronization
115.           private static Object mouseLock = new Object();
116.           private static Object keyLock = new Object();
117.
118.           // default font
```

```
119.        private static final Font DEFAULT_FONT = new Font("SansSerif", Font.PLAIN,
    16);
120.
121.        // current font
122.        private static Font font;
123.
124.        // double buffered graphics
125.        private static BufferedImage offscreenImage, onscreenImage;
126.        private static Graphics2D offscreen, onscreen;
127.
128.        // singleton for callbacks: avoids generation of extra .class files
129.        private static StdDraw std = new StdDraw();
130.
131.        // the frame for drawing to the screen
132.        private static JFrame frame;
133.
134.        // mouse state
135.        private static boolean mousePressed = false;
136.        private static double mouseX = 0;
137.        private static double mouseY = 0;
138.
139.        // keyboard state
140.        private static LinkedList<Character> keysTyped = new LinkedList<Character>();
141.
142.        // not instantiable
143.        private StdDraw() { }
144.
145.
146.        // static initializer
147.        static { init(); }
148.
149.        /**
150.         * Set the window size to w-by-h pixels.
151.         *
152.         * @param w the width as a number of pixels
153.         * @param h the height as a number of pixels
154.         * @throws a RunTimeException if the width or height is 0 or negative
155.         */
156.        public static void setCanvasSize(int w, int h) {
157.            if (w < 1 || h < 1) throw new RuntimeException("width and height must be
    positive");
158.            width = w;
159.            height = h;
160.            init();
161.        }
162.
163.        // init
164.        private static void init() {
```

```
165.            if (frame != null) frame.setVisible(false);
166.            frame = new JFrame();
167.            offscreenImage = new BufferedImage(width, height,
    BufferedImage.TYPE_INT_ARGB);
168.            onscreenImage  = new BufferedImage(width, height,
    BufferedImage.TYPE_INT_ARGB);
169.            offscreen = offscreenImage.createGraphics();
170.            onscreen  = onscreenImage.createGraphics();
171.            setXscale();
172.            setYscale();
173.            offscreen.setColor(DEFAULT_CLEAR_COLOR);
174.            offscreen.fillRect(0, 0, width, height);
175.            setPenColor();
176.            setPenRadius();
177.            setFont();
178.            clear();
179.
180.            // add antialiasing
181.            RenderingHints hints = new
    RenderingHints(RenderingHints.KEY_ANTIALIASING,
182.                                           RenderingHints.VALUE_ANTIALIAS_ON);
183.            hints.put(RenderingHints.KEY_RENDERING,
    RenderingHints.VALUE_RENDER_QUALITY);
184.            offscreen.addRenderingHints(hints);
185.
186.            // frame stuff
187.            ImageIcon icon = new ImageIcon(onscreenImage);
188.            JLabel draw = new JLabel(icon);
189.
190.            draw.addMouseListener(std);
191.            draw.addMouseMotionListener(std);
192.
193.            frame.setContentPane(draw);
194.            frame.addKeyListener(std);    // JLabel cannot get keyboard focus
195.            frame.setResizable(false);
196.            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);           // closes all
    windows
197.            // frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);     // closes
    only current window
198.            frame.setTitle("Standard Draw");
199.            frame.setJMenuBar(createMenuBar());
200.            frame.pack();
201.            frame.requestFocusInWindow();
202.            frame.setVisible(true);
203.        }
204.
205.        // create the menu bar (changed to private)
206.        private static JMenuBar createMenuBar() {
```

```
207.            JMenuBar menuBar = new JMenuBar();
208.            JMenu menu = new JMenu("File");
209.            menuBar.add(menu);
210.            JMenuItem menuItem1 = new JMenuItem(" Save...   ");
211.            menuItem1.addActionListener(std);
212.            menuItem1.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_S,
213.                              Toolkit.getDefaultToolkit().getMenuShortcutKeyMask()));
214.            menu.add(menuItem1);
215.            return menuBar;
216.         }
217.
218.
219.      /************************************************************************
220.         *  User and screen coordinate systems
221.      ************************************************************************/
222.
223.         /**
224.          * Set the x-scale to be the default (between 0.0 and 1.0).
225.          */
226.         public static void setXscale() { setXscale(DEFAULT_XMIN, DEFAULT_XMAX);
      }
227.
228.         /**
229.          * Set the y-scale to be the default (between 0.0 and 1.0).
230.          */
231.         public static void setYscale() { setYscale(DEFAULT_YMIN, DEFAULT_YMAX);
      }
232.
233.         /**
234.          * Set the x-scale (a 10% border is added to the values)
235.          * @param min the minimum value of the x-scale
236.          * @param max the maximum value of the x-scale
237.          */
238.         public static void setXscale(double min, double max) {
239.            double size = max - min;
240.            xmin = min - BORDER * size;
241.            xmax = max + BORDER * size;
242.         }
243.
244.         /**
245.          * Set the y-scale (a 10% border is added to the values).
246.          * @param min the minimum value of the y-scale
247.          * @param max the maximum value of the y-scale
248.          */
249.         public static void setYscale(double min, double max) {
250.            double size = max - min;
```

```
251.             ymin = min - BORDER * size;
252.             ymax = max + BORDER * size;
253.          }
254.
255.          // helper functions that scale from user coordinates to screen coordinates and back
256.          private static double  scaleX(double x) { return width  * (x - xmin) / (xmax - xmin);
    }
257.          private static double  scaleY(double y) { return height * (ymax - y) / (ymax - ymin);
    }
258.          private static double factorX(double w) { return w * width  / Math.abs(xmax -
    xmin);  }
259.          private static double factorY(double h) { return h * height / Math.abs(ymax - ymin);
    }
260.          private static double   userX(double x) { return xmin + x * (xmax - xmin) / width;
    }
261.          private static double   userY(double y) { return ymax - y * (ymax - ymin) / height;
    }
262.
263.
264.          /**
265.           * Clear the screen to the default color (white).
266.           */
267.          public static void clear() { clear(DEFAULT_CLEAR_COLOR); }
268.          /**
269.           * Clear the screen to the given color.
270.           * @param color the Color to make the background
271.           */
272.          public static void clear(Color color) {
273.             offscreen.setColor(color);
274.             offscreen.fillRect(0, 0, width, height);
275.             offscreen.setColor(penColor);
276.             draw();
277.          }
278.
279.          /**
280.           * Get the current pen radius.
281.           */
282.          public static double getPenRadius() { return penRadius; }
283.
284.          /**
285.           * Set the pen size to the default (.002).
286.           */
287.          public static void setPenRadius() { setPenRadius(DEFAULT_PEN_RADIUS); }
288.          /**
289.           * Set the radius of the pen to the given size.
290.           * @param r the radius of the pen
291.           * @throws RuntimeException if r is negative
292.           */
```

```
293.          public static void setPenRadius(double r) {
294.             if (r < 0) throw new RuntimeException("pen radius must be positive");
295.             penRadius = r * DEFAULT_SIZE;
296.             BasicStroke stroke = new BasicStroke((float) penRadius,
    BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND);
297.             // BasicStroke stroke = new BasicStroke((float) penRadius);
298.             offscreen.setStroke(stroke);
299.          }
300.
301.          /**
302.           * Get the current pen color.
303.           */
304.          public static Color getPenColor() { return penColor; }
305.
306.          /**
307.           * Set the pen color to the default color (black).
308.           */
309.          public static void setPenColor() { setPenColor(DEFAULT_PEN_COLOR); }
310.          /**
311.           * Set the pen color to the given color. The available pen colors are
312.           * BLACK, BLUE, CYAN, DARK_GRAY, GRAY, GREEN, LIGHT_GRAY,
    MAGENTA,
313.           * ORANGE, PINK, RED, WHITE, and YELLOW.
314.           * @param color the Color to make the pen
315.           */
316.          public static void setPenColor(Color color) {
317.             penColor = color;
318.             offscreen.setColor(penColor);
319.          }
320.
321.          /**
322.           * Get the current font.
323.           */
324.          public static Font getFont() { return font; }
325.
326.          /**
327.           * Set the font to the default font (sans serif, 16 point).
328.           */
329.          public static void setFont() { setFont(DEFAULT_FONT); }
330.
331.          /**
332.           * Set the font to the given value.
333.           * @param f the font to make text
334.           */
335.          public static void setFont(Font f) { font = f; }
336.
337.
```

```
338.
    /*****************************************************************************
339.          *  Drawing geometric shapes.
340.
    *****************************************************************************/
341.
342.          /**
343.           * Draw a line from (x0, y0) to (x1, y1).
344.           * @param x0 the x-coordinate of the starting point
345.           * @param y0 the y-coordinate of the starting point
346.           * @param x1 the x-coordinate of the destination point
347.           * @param y1 the y-coordinate of the destination point
348.           */
349.          public static void line(double x0, double y0, double x1, double y1) {
350.             offscreen.draw(new Line2D.Double(scaleX(x0), scaleY(y0), scaleX(x1),
    scaleY(y1)));
351.             draw();
352.          }
353.
354.          /**
355.           * Draw one pixel at (x, y).
356.           * @param x the x-coordinate of the pixel
357.           * @param y the y-coordinate of the pixel
358.           */
359.          private static void pixel(double x, double y) {
360.             offscreen.fillRect((int) Math.round(scaleX(x)), (int) Math.round(scaleY(y)), 1,
    1);
361.          }
362.
363.          /**
364.           * Draw a point at (x, y).
365.           * @param x the x-coordinate of the point
366.           * @param y the y-coordinate of the point
367.           */
368.          public static void point(double x, double y) {
369.             double xs = scaleX(x);
370.             double ys = scaleY(y);
371.             double r = penRadius;
372.             // double ws = factorX(2*r);
373.             // double hs = factorY(2*r);
374.             // if (ws <= 1 && hs <= 1) pixel(x, y);
375.             if (r <= 1) pixel(x, y);
376.             else offscreen.fill(new Ellipse2D.Double(xs - r/2, ys - r/2, r, r));
377.             draw();
378.          }
379.
380.          /**
381.           * Draw a circle of radius r, centered on (x, y).
```

```
382.          * @param x the x-coordinate of the center of the circle
383.          * @param y the y-coordinate of the center of the circle
384.          * @param r the radius of the circle
385.          * @throws RuntimeException if the radius of the circle is negative
386.          */
387.         public static void circle(double x, double y, double r) {
388.            if (r < 0) throw new RuntimeException("circle radius can't be negative");
389.            double xs = scaleX(x);
390.            double ys = scaleY(y);
391.            double ws = factorX(2*r);
392.            double hs = factorY(2*r);
393.            if (ws <= 1 && hs <= 1) pixel(x, y);
394.            else offscreen.draw(new Ellipse2D.Double(xs - ws/2, ys - hs/2, ws, hs));
395.            draw();
396.         }
397.
398.         /**
399.          * Draw filled circle of radius r, centered on (x, y).
400.          * @param x the x-coordinate of the center of the circle
401.          * @param y the y-coordinate of the center of the circle
402.          * @param r the radius of the circle
403.          * @throws RuntimeException if the radius of the circle is negative
404.          */
405.         public static void filledCircle(double x, double y, double r) {
406.            if (r < 0) throw new RuntimeException("circle radius can't be negative");
407.            double xs = scaleX(x);
408.            double ys = scaleY(y);
409.            double ws = factorX(2*r);
410.            double hs = factorX(2*r);
411.            if (ws <= 1 && hs <= 1) pixel(x, y);
412.            else offscreen.fill(new Ellipse2D.Double(xs - ws/2, ys - hs/2, ws, hs));
413.            draw();
414.         }
415.
416.
417.         /**
418.          * Draw an ellipse with given semimajor and semiminor axes, centered on (x, y).
419.          * @param x the x-coordinate of the center of the ellipse
420.          * @param y the y-coordinate of the center of the ellipse
421.          * @param semiMajorAxis is the semimajor axis of the ellipse
422.          * @param semiMinorAxis is the semiminor axis of the ellipse
423.          * @throws RuntimeException if either of the axes are negative
424.          */
425.         public static void ellipse(double x, double y, double semiMajorAxis, double
    semiMinorAxis) {
426.            if (semiMajorAxis < 0) throw new RuntimeException("ellipse semimajor axis
    can't be negative");
```

```
427.            if (semiMinorAxis < 0) throw new RuntimeException("ellipse semiminor axis
    can't be negative");
428.            double xs = scaleX(x);
429.            double ys = scaleY(y);
430.            double ws = factorX(2*semiMajorAxis);
431.            double hs = factorY(2*semiMinorAxis);
432.            if (ws <= 1 && hs <= 1) pixel(x, y);
433.            else offscreen.draw(new Ellipse2D.Double(xs - ws/2, ys - hs/2, ws, hs));
434.            draw();
435.        }
436.
437.        /**
438.         * Draw an ellipse with given semimajor and semiminor axes, centered on (x, y).
439.         * @param x the x-coordinate of the center of the ellipse
440.         * @param y the y-coordinate of the center of the ellipse
441.         * @param semiMajorAxis is the semimajor axis of the ellipse
442.         * @param semiMinorAxis is the semiminor axis of the ellipse
443.         * @throws RuntimeException if either of the axes are negative
444.         */
445.        public static void filledEllipse(double x, double y, double semiMajorAxis, double
    semiMinorAxis) {
446.            if (semiMajorAxis < 0) throw new RuntimeException("ellipse semimajor axis
    can't be negative");
447.            if (semiMinorAxis < 0) throw new RuntimeException("ellipse semiminor axis
    can't be negative");
448.            double xs = scaleX(x);
449.            double ys = scaleY(y);
450.            double ws = factorX(2*semiMajorAxis);
451.            double hs = factorY(2*semiMinorAxis);
452.            if (ws <= 1 && hs <= 1) pixel(x, y);
453.            else offscreen.fill(new Ellipse2D.Double(xs - ws/2, ys - hs/2, ws, hs));
454.            draw();
455.        }
456.
457.
458.        /**
459.         * Draw an arc of radius r, centered on (x, y), from angle1 to angle2 (in degrees).
460.         * @param x the x-coordinate of the center of the circle
461.         * @param y the y-coordinate of the center of the circle
462.         * @param r the radius of the circle
463.         * @param angle1 the starting angle. 0 would mean an arc beginning at 3 o'clock.
464.         * @param angle2 the angle at the end of the arc. For example, if
465.         *       you want a 90 degree arc, then angle2 should be angle1 + 90.
466.         * @throws RuntimeException if the radius of the circle is negative
467.         */
468.        public static void arc(double x, double y, double r, double angle1, double angle2) {
469.            if (r < 0) throw new RuntimeException("arc radius can't be negative");
470.            while (angle2 < angle1) angle2 += 360;
```

```
471.            double xs = scaleX(x);
472.            double ys = scaleY(y);
473.            double ws = factorX(2*r);
474.            double hs = factorY(2*r);
475.            if (ws <= 1 && hs <= 1) pixel(x, y);
476.            else offscreen.draw(new Arc2D.Double(xs - ws/2, ys - hs/2, ws, hs, angle1,
    angle2 - angle1, Arc2D.OPEN));
477.            draw();
478.         }
479.
480.         /**
481.          * Draw a square of side length 2r, centered on (x, y).
482.          * @param x the x-coordinate of the center of the square
483.          * @param y the y-coordinate of the center of the square
484.          * @param r radius is half the length of any side of the square
485.          * @throws RuntimeException if r is negative
486.          */
487.         public static void square(double x, double y, double r) {
488.            if (r < 0) throw new RuntimeException("square side length can't be negative");
489.            double xs = scaleX(x);
490.            double ys = scaleY(y);
491.            double ws = factorX(2*r);
492.            double hs = factorY(2*r);
493.            if (ws <= 1 && hs <= 1) pixel(x, y);
494.            else offscreen.draw(new Rectangle2D.Double(xs - ws/2, ys - hs/2, ws, hs));
495.            draw();
496.         }
497.
498.         /**
499.          * Draw a filled square of side length 2r, centered on (x, y).
500.          * @param x the x-coordinate of the center of the square
501.          * @param y the y-coordinate of the center of the square
502.          * @param r radius is half the length of any side of the square
503.          * @throws RuntimeException if r is negative
504.          */
505.         public static void filledSquare(double x, double y, double r) {
506.            if (r < 0) throw new RuntimeException("square side length can't be negative");
507.            double xs = scaleX(x);
508.            double ys = scaleY(y);
509.            double ws = factorX(2*r);
510.            double hs = factorY(2*r);
511.            if (ws <= 1 && hs <= 1) pixel(x, y);
512.            else offscreen.fill(new Rectangle2D.Double(xs - ws/2, ys - hs/2, ws, hs));
513.            draw();
514.         }
515.
516.
517.         /**
```

```
518.          * Draw a rectangle of given half width and half height, centered on (x, y).
519.          * @param x the x-coordinate of the center of the rectangle
520.          * @param y the y-coordinate of the center of the rectangle
521.          * @param halfWidth is half the width of the rectangle
522.          * @param halfHeight is half the height of the rectangle
523.          * @throws RuntimeException if halfWidth or halfHeight is negative
524.          */
525.         public static void rectangle(double x, double y, double halfWidth, double
    halfHeight) {
526.             if (halfWidth  < 0) throw new RuntimeException("half width can't be negative");
527.             if (halfHeight < 0) throw new RuntimeException("half height can't be negative");
528.             double xs = scaleX(x);
529.             double ys = scaleY(y);
530.             double ws = factorX(2*halfWidth);
531.             double hs = factorY(2*halfHeight);
532.             if (ws <= 1 && hs <= 1) pixel(x, y);
533.             else offscreen.draw(new Rectangle2D.Double(xs - ws/2, ys - hs/2, ws, hs));
534.             draw();
535.         }
536.
537.         /**
538.          * Draw a filled rectangle of given half width and half height, centered on (x, y).
539.          * @param x the x-coordinate of the center of the rectangle
540.          * @param y the y-coordinate of the center of the rectangle
541.          * @param halfWidth is half the width of the rectangle
542.          * @param halfHeight is half the height of the rectangle
543.          * @throws RuntimeException if halfWidth or halfHeight is negative
544.          */
545.         public static void filledRectangle(double x, double y, double halfWidth, double
    halfHeight) {
546.             if (halfWidth  < 0) throw new RuntimeException("half width can't be negative");
547.             if (halfHeight < 0) throw new RuntimeException("half height can't be negative");
548.             double xs = scaleX(x);
549.             double ys = scaleY(y);
550.             double ws = factorX(2*halfWidth);
551.             double hs = factorY(2*halfHeight);
552.             if (ws <= 1 && hs <= 1) pixel(x, y);
553.             else offscreen.fill(new Rectangle2D.Double(xs - ws/2, ys - hs/2, ws, hs));
554.             draw();
555.         }
556.
557.
558.         /**
559.          * Draw a polygon with the given (x[i], y[i]) coordinates.
560.          * @param x an array of all the x-coordindates of the polygon
561.          * @param y an array of all the y-coordindates of the polygon
562.          */
563.         public static void polygon(double[] x, double[] y) {
```

```
564.            int N = x.length;
565.            GeneralPath path = new GeneralPath();
566.            path.moveTo((float) scaleX(x[0]), (float) scaleY(y[0]));
567.            for (int i = 0; i < N; i++)
568.               path.lineTo((float) scaleX(x[i]), (float) scaleY(y[i]));
569.            path.closePath();
570.            offscreen.draw(path);
571.            draw();
572.         }
573.
574.         /**
575.          * Draw a filled polygon with the given (x[i], y[i]) coordinates.
576.          * @param x an array of all the x-coordindates of the polygon
577.          * @param y an array of all the y-coordindates of the polygon
578.          */
579.         public static void filledPolygon(double[] x, double[] y) {
580.            int N = x.length;
581.            GeneralPath path = new GeneralPath();
582.            path.moveTo((float) scaleX(x[0]), (float) scaleY(y[0]));
583.            for (int i = 0; i < N; i++)
584.               path.lineTo((float) scaleX(x[i]), (float) scaleY(y[i]));
585.            path.closePath();
586.            offscreen.fill(path);
587.            draw();
588.         }
589.
590.
591.
592.      /*************************************************************************
593.         *  Drawing images.
594.      *************************************************************************/
595.
596.         // get an image from the given filename
597.         private static Image getImage(String filename) {
598.
599.            // to read from file
600.            ImageIcon icon = new ImageIcon(filename);
601.
602.            // try to read from URL
603.            if ((icon == null) || (icon.getImageLoadStatus() != MediaTracker.COMPLETE))
      {
604.               try {
605.                  URL url = new URL(filename);
606.                  icon = new ImageIcon(url);
607.               } catch (Exception e) { /* not a url */ }
608.            }
```

```
609.
610.            // in case file is inside a .jar
611.            if ((icon == null) || (icon.getImageLoadStatus() != MediaTracker.COMPLETE))
    {
612.               URL url = StdDraw.class.getResource(filename);
613.               if (url == null) throw new RuntimeException("image " + filename + " not
    found");
614.               icon = new ImageIcon(url);
615.            }
616.
617.            return icon.getImage();
618.         }
619.
620.         /**
621.          * Draw picture (gif, jpg, or png) centered on (x, y).
622.          * @param x the center x-coordinate of the image
623.          * @param y the center y-coordinate of the image
624.          * @param s the name of the image/picture, e.g., "ball.gif"
625.          * @throws RuntimeException if the image is corrupt
626.          */
627.         public static void picture(double x, double y, String s) {
628.            Image image = getImage(s);
629.            double xs = scaleX(x);
630.            double ys = scaleY(y);
631.            int ws = image.getWidth(null);
632.            int hs = image.getHeight(null);
633.            if (ws < 0 || hs < 0) throw new RuntimeException("image " + s + " is corrupt");
634.
635.            offscreen.drawImage(image, (int) Math.round(xs - ws/2.0), (int) Math.round(ys -
    hs/2.0), null);
636.            draw();
637.         }
638.
639.         /**
640.          * Draw picture (gif, jpg, or png) centered on (x, y),
641.          * rotated given number of degrees
642.          * @param x the center x-coordinate of the image
643.          * @param y the center y-coordinate of the image
644.          * @param s the name of the image/picture, e.g., "ball.gif"
645.          * @param degrees is the number of degrees to rotate counterclockwise
646.          * @throws RuntimeException if the image is corrupt
647.          */
648.         public static void picture(double x, double y, String s, double degrees) {
649.            Image image = getImage(s);
650.            double xs = scaleX(x);
651.            double ys = scaleY(y);
652.            int ws = image.getWidth(null);
653.            int hs = image.getHeight(null);
```

```
654.            if (ws < 0 || hs < 0) throw new RuntimeException("image " + s + " is corrupt");
655.
656.            offscreen.rotate(Math.toRadians(-degrees), xs, ys);
657.            offscreen.drawImage(image, (int) Math.round(xs - ws/2.0), (int) Math.round(ys -
    hs/2.0), null);
658.            offscreen.rotate(Math.toRadians(+degrees), xs, ys);
659.
660.            draw();
661.         }
662.
663.         /**
664.          * Draw picture (gif, jpg, or png) centered on (x, y), rescaled to w-by-h.
665.          * @param x the center x coordinate of the image
666.          * @param y the center y coordinate of the image
667.          * @param s the name of the image/picture, e.g., "ball.gif"
668.          * @param w the width of the image
669.          * @param h the height of the image
670.          * @throws RuntimeException if the width height are negative
671.          * @throws RuntimeException if the image is corrupt
672.          */
673.         public static void picture(double x, double y, String s, double w, double h) {
674.            Image image = getImage(s);
675.            double xs = scaleX(x);
676.            double ys = scaleY(y);
677.            if (w < 0) throw new RuntimeException("width is negative: " + w);
678.            if (h < 0) throw new RuntimeException("height is negative: " + h);
679.            double ws = factorX(w);
680.            double hs = factorY(h);
681.            if (ws < 0 || hs < 0) throw new RuntimeException("image " + s + " is corrupt");
682.            if (ws <= 1 && hs <= 1) pixel(x, y);
683.            else {
684.               offscreen.drawImage(image, (int) Math.round(xs - ws/2.0),
685.                                          (int) Math.round(ys - hs/2.0),
686.                                          (int) Math.round(ws),
687.                                          (int) Math.round(hs), null);
688.            }
689.            draw();
690.         }
691.
692.
693.         /**
694.          * Draw picture (gif, jpg, or png) centered on (x, y), rotated
695.          * given number of degrees, rescaled to w-by-h.
696.          * @param x the center x-coordinate of the image
697.          * @param y the center y-coordinate of the image
698.          * @param s the name of the image/picture, e.g., "ball.gif"
699.          * @param w the width of the image
700.          * @param h the height of the image
```

```
701.          * @param degrees is the number of degrees to rotate counterclockwise
702.          * @throws RuntimeException if the image is corrupt
703.          */
704.         public static void picture(double x, double y, String s, double w, double h, double
    degrees) {
705.            Image image = getImage(s);
706.            double xs = scaleX(x);
707.            double ys = scaleY(y);
708.            double ws = factorX(w);
709.            double hs = factorY(h);
710.            if (ws < 0 || hs < 0) throw new RuntimeException("image " + s + " is corrupt");
711.            if (ws <= 1 && hs <= 1) pixel(x, y);
712.
713.            offscreen.rotate(Math.toRadians(-degrees), xs, ys);
714.            offscreen.drawImage(image, (int) Math.round(xs - ws/2.0),
715.                                       (int) Math.round(ys - hs/2.0),
716.                                       (int) Math.round(ws),
717.                                       (int) Math.round(hs), null);
718.            offscreen.rotate(Math.toRadians(+degrees), xs, ys);
719.
720.            draw();
721.         }
722.
723.
724.
    /***************************************************************************
725.          *  Drawing text.
726.
    ***************************************************************************/
727.
728.         /**
729.          * Write the given text string in the current font, centered on (x, y).
730.          * @param x the center x-coordinate of the text
731.          * @param y the center y-coordinate of the text
732.          * @param s the text
733.          */
734.         public static void text(double x, double y, String s) {
735.            offscreen.setFont(font);
736.            FontMetrics metrics = offscreen.getFontMetrics();
737.            double xs = scaleX(x);
738.            double ys = scaleY(y);
739.            int ws = metrics.stringWidth(s);
740.            int hs = metrics.getDescent();
741.            offscreen.drawString(s, (float) (xs - ws/2.0), (float) (ys + hs));
742.            draw();
743.         }
744.
745.         /**
```

```
746.          * Write the given text string in the current font, centered on (x, y) and
747.          * rotated by the specified number of degrees
748.          * @param x the center x-coordinate of the text
749.          * @param y the center y-coordinate of the text
750.          * @param s the text
751.          * @param degrees is the number of degrees to rotate counterclockwise
752.          */
753.         public static void text(double x, double y, String s, double degrees) {
754.            double xs = scaleX(x);
755.            double ys = scaleY(y);
756.            offscreen.rotate(Math.toRadians(-degrees), xs, ys);
757.            text(x, y, s);
758.            offscreen.rotate(Math.toRadians(+degrees), xs, ys);
759.         }
760.
761.
762.         /**
763.          * Write the given text string in the current font, left-aligned at (x, y).
764.          * @param x the x-coordinate of the text
765.          * @param y the y-coordinate of the text
766.          * @param s the text
767.          */
768.         public static void textLeft(double x, double y, String s) {
769.            offscreen.setFont(font);
770.            FontMetrics metrics = offscreen.getFontMetrics();
771.            double xs = scaleX(x);
772.            double ys = scaleY(y);
773.            int ws = metrics.stringWidth(s);
774.            int hs = metrics.getDescent();
775.            offscreen.drawString(s, (float) (xs), (float) (ys + hs));
776.            show();
777.         }
778.
779.         /**
780.          * Write the given text string in the current font, right-aligned at (x, y).
781.          * @param x the x-coordinate of the text
782.          * @param y the y-coordinate of the text
783.          * @param s the text
784.          */
785.         public static void textRight(double x, double y, String s) {
786.            offscreen.setFont(font);
787.            FontMetrics metrics = offscreen.getFontMetrics();
788.            double xs = scaleX(x);
789.            double ys = scaleY(y);
790.            int ws = metrics.stringWidth(s);
791.            int hs = metrics.getDescent();
792.            offscreen.drawString(s, (float) (xs - ws), (float) (ys + hs));
793.            show();
```

```
794.        }
795.
796.
797.
798.        /**
799.         * Display on screen, pause for t milliseconds, and turn on
800.         * <em>animation mode</em>: subsequent calls to
801.         * drawing methods such as <tt>line()</tt>, <tt>circle()</tt>, and <tt>square()</tt>
802.         * will not be displayed on screen until the next call to <tt>show()</tt>.
803.         * This is useful for producing animations (clear the screen, draw a bunch of shapes,
804.         * display on screen for a fixed amount of time, and repeat). It also speeds up
805.         * drawing a huge number of shapes (call <tt>show(0)</tt> to defer drawing
806.         * on screen, draw the shapes, and call <tt>show(0)</tt> to display them all
807.         * on screen at once).
808.         * @param t number of milliseconds
809.         */
810.        public static void show(int t) {
811.           defer = false;
812.           draw();
813.           try { Thread.currentThread().sleep(t); }
814.           catch (InterruptedException e) { System.out.println("Error sleeping"); }
815.           defer = true;
816.        }
817.
818.
819.        /**
820.         * Display on-screen and turn off animation mode:
821.         * subsequent calls to
822.         * drawing methods such as <tt>line()</tt>, <tt>circle()</tt>, and <tt>square()</tt>
823.         * will be displayed on screen when called. This is the default.
824.         */
825.        public static void show() {
826.           defer = false;
827.           draw();
828.        }
829.
830.        // draw onscreen if defer is false
831.        private static void draw() {
832.           if (defer) return;
833.           onscreen.drawImage(offscreenImage, 0, 0, null);
834.           frame.repaint();
835.        }
836.
837.
838.     /*************************************************************************
839.        *  Save drawing to a file.
```

```
840.
    ***************************************************************************/
841.
842.        /**
843.         * Save to file - suffix must be png, jpg, or gif.
844.         * @param filename the name of the file with one of the required suffixes
845.         */
846.        public static void save(String filename) {
847.           File file = new File(filename);
848.           String suffix = filename.substring(filename.lastIndexOf('.') + 1);
849.
850.           // png files
851.           if (suffix.toLowerCase().equals("png")) {
852.              try { ImageIO.write(offscreenImage, suffix, file); }
853.              catch (IOException e) { e.printStackTrace(); }
854.           }
855.
856.           // need to change from ARGB to RGB for jpeg
857.           // reference: http://archives.java.sun.com/cgi-bin/wa?A2=ind0404&L=java2d-
    interest&D=0&P=2727
858.           else if (suffix.toLowerCase().equals("jpg")) {
859.              WritableRaster raster = offscreenImage.getRaster();
860.              WritableRaster newRaster;
861.              newRaster = raster.createWritableChild(0, 0, width, height, 0, 0, new int[] {0,
    1, 2});
862.              DirectColorModel cm = (DirectColorModel)
    offscreenImage.getColorModel();
863.              DirectColorModel newCM = new DirectColorModel(cm.getPixelSize(),
864.                                                    cm.getRedMask(),
865.                                                    cm.getGreenMask(),
866.                                                    cm.getBlueMask());
867.              BufferedImage rgbBuffer = new BufferedImage(newCM, newRaster, false,
    null);
868.              try { ImageIO.write(rgbBuffer, suffix, file); }
869.              catch (IOException e) { e.printStackTrace(); }
870.           }
871.
872.           else {
873.              System.out.println("Invalid image file type: " + suffix);
874.           }
875.        }
876.
877.
878.        /**
879.         * This method cannot be called directly.
880.         */
881.        public void actionPerformed(ActionEvent e) {
```

```
882.              FileDialog chooser = new FileDialog(StdDraw.frame, "Use a .png or .jpg
    extension", FileDialog.SAVE);
883.              chooser.setVisible(true);
884.              String filename = chooser.getFile();
885.              if (filename != null) {
886.                 StdDraw.save(chooser.getDirectory() + File.separator + chooser.getFile());
887.              }
888.           }
889.
890.
891.
    /*************************************************************************
892.           *  Mouse interactions.
893.
    *************************************************************************/
894.
895.           /**
896.            * Is the mouse being pressed?
897.            * @return true or false
898.            */
899.           public static boolean mousePressed() {
900.              synchronized (mouseLock) {
901.                 return mousePressed;
902.              }
903.           }
904.
905.           /**
906.            * What is the x-coordinate of the mouse?
907.            * @return the value of the x-coordinate of the mouse
908.            */
909.           public static double mouseX() {
910.              synchronized (mouseLock) {
911.                 return mouseX;
912.              }
913.           }
914.
915.           /**
916.            * What is the y-coordinate of the mouse?
917.            * @return the value of the y-coordinate of the mouse
918.            */
919.           public static double mouseY() {
920.              synchronized (mouseLock) {
921.                 return mouseY;
922.              }
923.           }
924.
925.
926.           /**
```

```
927.          * This method cannot be called directly.
928.          */
929.         public void mouseClicked(MouseEvent e) { }
930.
931.         /**
932.          * This method cannot be called directly.
933.          */
934.         public void mouseEntered(MouseEvent e) { }
935.
936.         /**
937.          * This method cannot be called directly.
938.          */
939.         public void mouseExited(MouseEvent e) { }
940.
941.         /**
942.          * This method cannot be called directly.
943.          */
944.         public void mousePressed(MouseEvent e) {
945.            synchronized (mouseLock) {
946.               mouseX = StdDraw.userX(e.getX());
947.               mouseY = StdDraw.userY(e.getY());
948.               mousePressed = true;
949.            }
950.         }
951.
952.         /**
953.          * This method cannot be called directly.
954.          */
955.         public void mouseReleased(MouseEvent e) {
956.            synchronized (mouseLock) {
957.               mousePressed = false;
958.            }
959.         }
960.
961.         /**
962.          * This method cannot be called directly.
963.          */
964.         public void mouseDragged(MouseEvent e)  {
965.            synchronized (mouseLock) {
966.               mouseX = StdDraw.userX(e.getX());
967.               mouseY = StdDraw.userY(e.getY());
968.            }
969.         }
970.
971.         /**
972.          * This method cannot be called directly.
973.          */
974.         public void mouseMoved(MouseEvent e) {
```

```
975.            synchronized (mouseLock) {
976.               mouseX = StdDraw.userX(e.getX());
977.               mouseY = StdDraw.userY(e.getY());
978.            }
979.         }
980.
981.
982.
    /*****************************************************************************
983.         *  Keyboard interactions.
984.
    *****************************************************************************/
985.
986.         /**
987.          * Has the user typed a key?
988.          * @return true if the user has typed a key, false otherwise
989.          */
990.         public static boolean hasNextKeyTyped() {
991.            synchronized (keyLock) {
992.               return !keysTyped.isEmpty();
993.            }
994.         }
995.
996.         /**
997.          * What is the next key that was typed by the user?
998.          * @return the next key typed
999.          */
1000.        public static char nextKeyTyped() {
1001.           synchronized (keyLock) {
1002.              return keysTyped.removeLast();
1003.           }
1004.        }
1005.
1006.        /**
1007.         * This method cannot be called directly.
1008.         */
1009.        public void keyTyped(KeyEvent e) {
1010.           synchronized (keyLock) {
1011.              keysTyped.addFirst(e.getKeyChar());
1012.           }
1013.        }
1014.
1015.        /**
1016.         * This method cannot be called directly.
1017.         */
1018.        public void keyPressed(KeyEvent e) { }
1019.
1020.        /**
```

```
1021.          * This method cannot be called directly.
1022.          */
1023.         public void keyReleased(KeyEvent e) { }
1024.
1025.
1026.
1027.
1028.         /**
1029.          * Test client.
1030.          */
1031.         public static void main(String[] args) {
1032.            StdDraw.square(.2, .8, .1);
1033.            StdDraw.filledSquare(.8, .8, .2);
1034.            StdDraw.circle(.8, .2, .2);
1035.
1036.            StdDraw.setPenColor(StdDraw.BOOK_RED);
1037.            StdDraw.setPenRadius(.02);
1038.            StdDraw.arc(.8, .2, .1, 200, 45);
1039.
1040.            // draw a blue diamond
1041.            StdDraw.setPenRadius();
1042.            StdDraw.setPenColor(StdDraw.BOOK_BLUE);
1043.            double[] x = { .1, .2, .3, .2 };
1044.            double[] y = { .2, .3, .2, .1 };
1045.            StdDraw.filledPolygon(x, y);
1046.
1047.            // text
1048.            StdDraw.setPenColor(StdDraw.BLACK);
1049.            StdDraw.text(0.2, 0.5, "black text");
1050.            StdDraw.setPenColor(StdDraw.WHITE);
1051.            StdDraw.text(0.8, 0.8, "white text");
1052.         }
1053.
1054.      }
1055.
```

Appendix C.B - 2.DrawTour

```
1. import java.io.File;
2. import java.util.Scanner;
```

```
3.
4.  /**
5.   * Created by IntelliJ IDEA.
6.   * User: Asger
7.   * Date: 11-05-11
8.   * Time: 18:57
9.   * To change this template use File | Settings | File Templates.
10.  */
11. public class DrawTour {
12.
13. /* Written by Keld Helsgaun, November 2010 */
14.
15.    /** Draws a tour after a given time delay
16.     * @param t the tour (a permutation of the integers 0 to n-1)
17.     * @param x x-coordinates of the cities
18.     * @param y y-coordinates of the cities
19.     * @param delay the delay (in milliseconds)
20.     */
21.
22.
23.    public static void draw(int[] t, double[] x, double[] y, int delay) {
24.       int n = x.length;
25.          double xMin = x[0], xMax = x[0], yMin = y[0], yMax = y[0];
26.          for (int i = 1; i < n; i++) {
27.             xMin = Math.min(xMin, x[i]);
28.             xMax = Math.max(xMax, x[i]);
29.             yMin = Math.min(yMin, y[i]);
30.             yMax = Math.max(yMax, y[i]);
31.          }
32.          final int size = 800;
33.          int width = size;
34.          int height = (int) (width * (yMax - yMin) / (xMax - xMin));
35.          if (height > size) {
36.             width *= ((double) size) / height;
37.             height = size;
38.          }
39.          StdDraw.setCanvasSize(width, height);
40.          StdDraw.show(0);
41.          StdDraw.clear();
42.          StdDraw.setPenColor();
43.          StdDraw.setPenRadius();
44.          StdDraw.setXscale(xMin, xMax);
45.          StdDraw.setYscale(yMin, yMax);
46.          if (t != null) {
47.             for (int i = 1; i < t.length; i++)
48.                StdDraw.line(x[t[i - 1]], y[t[i - 1]], x[t[i]], y[t[i]]);
49.             StdDraw.line(x[t[n - 1]], y[t[n - 1]], x[t[0]], y[t[0]]);
50.          }
```

```
51.          StdDraw.setPenColor(StdDraw.RED);
52.          StdDraw.setPenRadius(0.005);
53.          for (int i = 0; i < n; i++)
54.             StdDraw.point(x[i], y[i]);
55.          StdDraw.show(delay);
56.    }
57.
58.
59.       /** Draws a tour without delay */
60.       public static void draw(int[]t, double[]x, double[]y) {
61.          draw(t, x, y, 0);
62.       }
63.
64.       /** Plots cities */
65.       public static void draw(double[]x, double[]y) {
66.          draw(null, x, y, 0);
67.       }
68.
69.       public static void main(String[]args) throws Exception {
70.          if (args.length != 2) {
71.             System.out.
72.                println("Usage: java DrawTour instance_file tour_file");
73.             System.exit(1);
74.          }
75.          Scanner instanceScanner = new Scanner(new File(args[0]));
76.          int n = instanceScanner.nextInt();
77.          double[] x = new double[n], y = new double[n];
78.          for (int i = 0; i < n; i++) {
79.             int j = instanceScanner.nextInt() - 1;
80.             /* Swap x and y for geographical instances */
81.             y[j] = instanceScanner.nextDouble();
82.             x[j] = instanceScanner.nextDouble();
83.          }
84.          Scanner tourScanner = new Scanner(new File(args[1]));
85.          int[] tour = new int[n];
86.          for (int i = 0; i < n; i++)
87.             tour[i] = tourScanner.nextInt() - 1;
88.          draw(tour, x, y, 0);
89.       }
90. }
```


Our program.