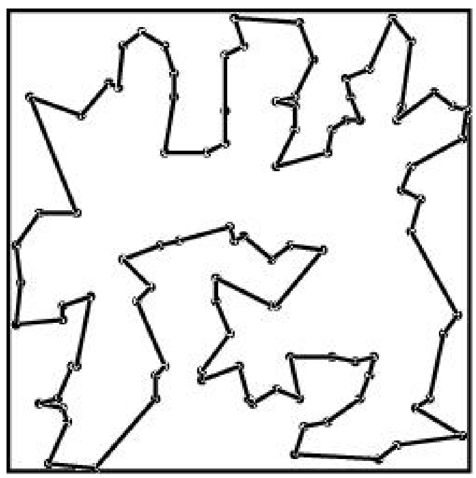
Roskilde University
Bachelor of Computer Sciences, 2nd module

ACO and TSP

29th of May, 2007

Supervisor:
Keld Helsgaun



Jean-Luc Ngassa
Jakob Kierkegaard

Abstract

We took an existing ant colony system framework with an accompanying TSP algorithm, which we changed by implementing different algorithms and extra functionality, in an attempt to acheive better tour constructions. We then extended it by implementing 2-opt and 2,5-opt optimization algorithms.

We tested the program with cases found on TSPLIB, and compared our results with results from the original application and a third party ant colony algorithm.

These results were then compared with the official optimal solutions. Even though our results never reached the optimal values, we achieved better results than those from the original framework without using our optimization algorithms, but they were never able to compete with the third party application. When applying optimization, we achieved results much better than the original framework, and slightly better than those gotten from the third party application.

Foreword

This report is part of the 2nd bachelor module at Computer Science at RUC.

We would like to thank our supervisor, Keld Helsgaun, for his unconditional support, his assistance and advice throughout the process.

We would also like to thank Jean-Luc's father Jean-Claude Njitche for his undirected support.

Unless otherwise stated, all figures have been made by us.

Footnotes are numbered continuously throughout the report. Tables and figures are numbered using the chapter number and the number the table or figure is in the current chapter. E.g. the second table in chapter 3 would have the number "3.2".

Jean-Luc Ngassa
Jakob Kierkegaard
Computer Sciences, RUC, 29th of May, 2007

Contents

1 Introduction

An initial look at ants in nature does not give an impression of an animal with a high IQ, but a closer look reveals that they are highly efficient in at least one task; finding the shortest route between two points. Starting from the hive they are prone to walk randomly around until they find a point of interest, e.g. a food source. When traveling back to the hive, they will deposit a chemical substance called *pheromone* as they go, which will help them find their way back to where they came from. When other ants encounter the path of pheromone, they will follow it, becoming less random in their movement. These will then also deposit pheromone, strengthening the already existing path. Because pheromone is a volatile substance, a constant stream of ants is required to keep up the strength of the trail. This means that if a shorter trail exists, the power of this trail's pheromone will be stronger, as the ants will traverse the trail in a shorter amount of time, while the pheromone still evaporates at the same speed. After a (relatively) short time span, the majority of the ants will therefore be following the shortest path, as this path has the strongest pheromone.

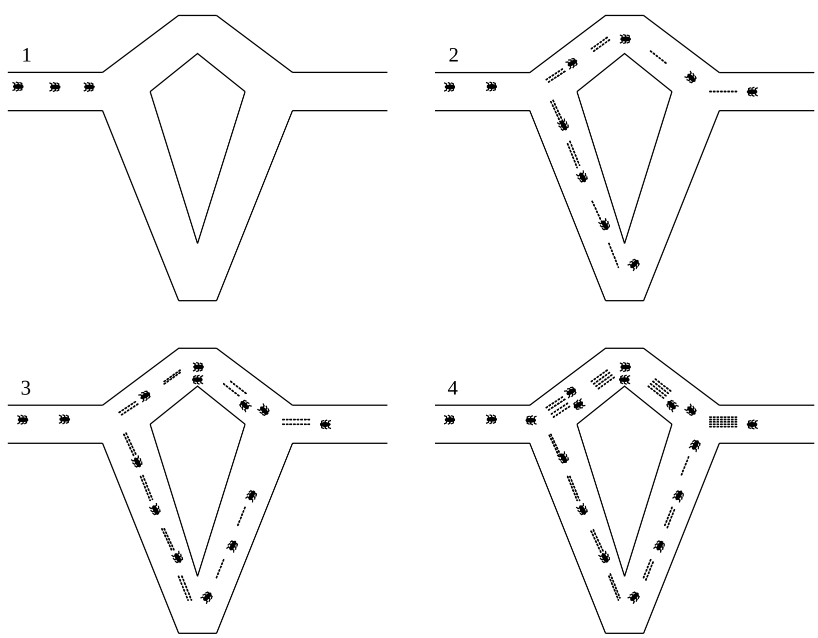


Figure 1.1: Ants encounter a split on their way to their food, and gradually they prioritize the shortest path because of the deployed pheromone (represented by the small lines).

An example of how the ants find the shortest way can be seen in figure 1.1, where the ants encounter a crossroad between their nest and the food source.

At first the chances to take either left or right are 50/50, but as the ants traverse the two distances, the pheromone increases faster on the shorter route and more ants end up taking that route.

Alberto Colorni, Vittorio Maniezzo and Marco Dorigo were the first to come up with an algorithm to simulate the ants' behavior, calling it an *Ant Colony System* (ACS) algorithm, commonly known as ACO; O for the optimizations that are made daily from the original ACS. As ACO contains general methods for solving various problems, it can be seen as a collection of several heuristic methods that can be applied to a large number of problems. ACO is part of the science "swarm intelligence", which is about using small independent units (e.g. bees, ants or birds to mention a few) to solve problems. Alone these units are dumb, but when they are allowed to communicate with each other and react to the responses, they are able to solve complex problems.

The problems that ACO can be applied to are too many to mention here, but one of the most popular ones is the *Traveling Salesman Problem*, known as the TSP. The TSP is the classical mathematical problem where a salesman has to pass through n cities (also called "nodes" as a more general term), and because he wants to complete the travel in the smallest amount of time, he needs to find out which route is the shortest. The TSP is known as a NP-hard problem, meaning that there is no deterministic polynomial computation to a final solution.

1.1 Motivation

When examining nature, people often find that animals and plants have evolved in an ingenious manner to survive. These new techniques can sometimes be used in modern day technology to solve problems that might be too time-consuming to solve by conservative means. After all, nature has had many millions of years to evolve into what it is now, whereas we have had less than 100 years to get technology to where it is now.

The concept of having something as simple as ants (or rather simulated ants) to solve a seemingly complex mathematical problem seemed interesting, and we wanted to find out if this method really was as good and functional as several sources claimed.

The purpose of our project is to design a framework for an ACO algorithms based on an existing one, that can be applied for constructing solutions for the TSP.

1.2 Problem Formulation

Ugo Chirico in [Chi04] developed a framework for ACO algorithms that was applied on the Travelling Salesman Problem and the Steiner Problem. His framework appeared to be quite excellent, but we also realised that the TSP implementation needed improvement, making us want to create a new and better framework and TSP implementation. During the design of the framework, we involved all aspects for constructing an ACO algorithm based mainly on [DS04]. Our main problem is:

> *How improved is the new framework compared to Chirico's, and how efficient is it at providing good results for the TSP compared to other TSP algorithms?*

Our level of efficiency is measured in accordance to the following settings:

- TSP tour solutions
- Time consumption

In order to approach the problem, we have split our problem definition into smaller problems:

- Tour Construction:
  Use Chirico's framework for Ant Colony Systems applied to the TSP, and make changes without ruining it's overall performance and usability as a framework.
- Tour Optimization:
  Extend the framework applied to the TSP so that it uses local search algorithms for improving retrieved tours from the ACO framework
- Visual Observation:
  Implement a GUI for tours' visualisation and a control panel for setting parameters.
- Experimental Observation:
  Change the values of the parameters for experimental analysis and general program evaluation.

1.3 Target audience

The target audience for this report is people who want to study the basics of ACO and how to apply it to a real life problem like TSP. Previous knowledge about ACO and TSP is not necessary, as we will introduce the reader to the necessary theory.

As this is a computer science project, it will be an advantage to have basic knowledge of the tools and methodologies used in this field of study. We will be doing all the programming in Java, so knowing the language will clearly be an advantage. But as the code is commented we believe that a person with experience in any object oriented programming language should be able to understand most of it.

As the project involves a fair amount of mathematical algorithms and modeling, experience in this field is recommended, as we do not include any chapters about the mathematical background theory.

1.4 Structure of the report

The report can be seen as having been split into 3 larger parts:

1. The first step contains the theory behind the practice, where we will indtroduce the ACO in chapter 2 and the TSP in chapter 3.
2. The second step is the optimization chapter describing options we have when wanting to optimize a tour which has been created by a TSP algorithm. This can be read in chapter 4.
3. The last step is the practical part of our report, involving the implementation as seen in chapter 5 and experiments as seen in chapter 6.

We will be rounding off with a discussion (chapter 7), conclusion (chapter 8) and finally the perspectives (chapter 9) of the report.

The bibliography will follow on page 47.

The appendix will contain screenshots (appendix A), and overview of the enclosed CD (appendix B). The code from Chirico's original program and our final application can lastly be seen in appendix D and E respectively.

2 An Ant Colony Framework

The design of ant colony algorithms is based on the search behavior of real ants. The ants' search behavior is based on a positive feedback from the cooperative behavior, based on the trail following of the other ants to reinforce good solutions on a problem. A solution for a shortest path problem is determined by the back and forth movements of ants on the path where shorter distances are more prioritized due to the higher concentration of pheromone.

2.1 Designing Ant Colony Algorithms

Let us consider an environment similar to the Double Bridge Experiment[1] where we have a colony of n ants traversing two branches AB and AC from A, the nest, to two food sources B and C. τAB and τAC are defined following a random distribution of a constant δ over [0,1].

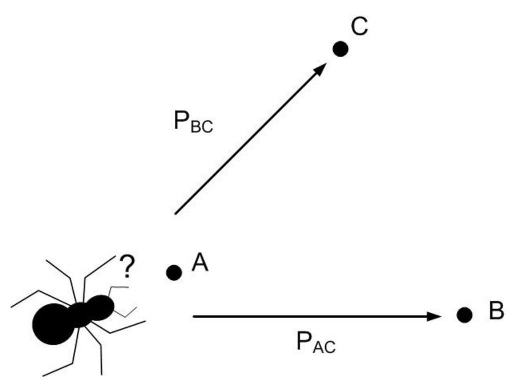


Figure 2.1: An ant's choice.

The random distribution is to give both branches a chance to be chosen by an ant, and it is between 0 and 1 as it is a random probability. With i as ant id, we define the following expression:

$$\tau AB_{i+1} = \begin{cases} \tau AB_i + 1, & \delta \leq P_{AB} \\ \tau AB_i, & \delta > P_{AB} \end{cases} \tag{2.1}$$

$$\tau AC_{i+1} = \begin{cases} \tau AC_i + 1, & \delta \leq P_{AC} \\ \tau AC_i, & \delta > P_{AC} \end{cases} \tag{2.2}$$

[1] The Double Bridge experiment is an experiment described in [GADP89] where the authors develop an understanding of the behaviour of ants.

The probability P_{AB} (resp. P_{AC}) is the probability that an ant chooses the city B (resp. C) as the next state position in the environment. It is enunciated by:

$$P_{AB} = \frac{\tau AB}{\tau AB + \tau AC} \tag{2.3}$$
$$(resp.\ P_{AC} = \frac{\tau AC}{\tau AB + \tau AC})$$

Remarks

- $P_{AB} + P_{AC} = 1$.
- It is obvious from equation 2.3 that the more ants traversing a branch, the greater the probability on that branch will be, and the higher are the chances for that branch to be chosen.

The traversal of a branch by an ant is equivalent to the deposit of pheromone, which makes the pheromone plays a big role in the choice of the moves of an ant in the environment. Therefore, an ant colony algorithm is an algorithm made on the basis of the pheromone trail and the state transition moves.

2.2 The Pheromone Trail Update

The pheromone biologically defines the modifications of the colony trail on the branches in the environment. As it is a volatile substance there is a time limit on its impact on the other ants. For such reasons, its computation is made under two considerations:

- The quantity of pheromone layed on a branch that has been used.
  Such quantity is expressed by two parameters, the parameter of deposit depending on the type of ants used for the simulation, and a parameter of decay for the deposit of pheromone evaluted as a probability between $[0, 1]$.The parameter of deposit is usually set proportionally to the inverted length of the branches traversed by the ant, so that short branches gets high pheromone deposit, simulating the environment described earlier.
- The quantity of pheromone evaporated after the ant has crossed a branch.
  The evaporation is set to control the evaporation on a path, based on the parameter of decay of the pheromone deposit by the previous ant(s).

However, for ant simulations and optimizations, there has been defined two rules for the pheromone update:

1. The local update
   The local update is the update of the pheromone on a single branch when it is traversed by an ant.

2. The global update
   The global update is the reinforcement of the branches in the best path found after each iteration of the ants in order to find the overall best path.

2.3 The State Transition Rule

The state transition rule is a set of rules that defines the next move of an artificial ant. Those rules are determined by using:

- The Constraint Satisfaction.
  This is a memory that helps ants to construct possible good solutions. As we have said when presenting ants, ants are chosing their path following a random probability. Constraint settings make an ant's choice of moves to be more constructive rather than a total dependance on a random choice.

- The Heuristic Desirability
  This is an evaluation of the closest steps, based on the inverted length on each branch. It is used to increase the amount of pheromone on small branches.

The state transition rule can be expressed as a random probability move function of the Heuristic Desirability and the Pheromone update. The transition rules are known as exploitation and exploration.

1. The exploitation rule.
   The exploitation rule is determined by the choice of edge with the highest amount of pheromone. It is straight forward using the heuristic desirability, the pheromone trail and several parameter settings.

2. The exploration rule
   The exploration rule is the search among all possible edges for the most probable edge that can be used to construct a good solution for the ant. However, such choice belongs to a probability that is set as well following parameters on ants' behaviors.

2.4 A Framework for an ant colony algorithm

An ant colony algorithm can be used for solving a huge variety of combinatorial problems as described in [DS04] following the above structure. However, the parameter settings of the pheromone trail update rule and the state transition rule depend on the problem the colony will have to solve, and the level of optimization the colony would have to perform. Building a framework is modelling a data-structure on a higher level abstraction that can be extended for solving such applications of the ant colony and/or used for other data-modelling. Following Dorigo in [DG97], Chirico in [Chi04] designed his framework as a distributed system of a colony of ants where ants perform the tasks described above; moving and updating the pherome in an environment represented by a graph. We have decided to keep the same general abstract design for two reasons:

1. A graph is a simplified representation of a physical or abstract environment using nodes and edges/cost.
   The different states an ant can move among in the environment are represented by the nodes in the graph, and the branches or arcs connecting

such nodes built by ants are edges. Edges can also be assimilated as cost, as it still involves the connection between two states.

2. A distributed system where each ant represent a thread and the colony is the shared object.
   The global update is assigned to the colony as it is the update that affects the behavior of the whole colony, while the local update is affected to the threaded ant, as such update is controlling the individual ant on its single meta-heuristic move.

The framework developed in [Chi04] was well structured in terms of classes and we ended up with the same classical construction:

- The Graph object
  This is the object defining the environment in which ants are capable of performing their moves.
- The Ant Colony object
  Ants are created using this object and run following a set amount of iterations. On an initial run, the pheromone is set on each path, and after each iteration, only the best path is being updated.
- The Ant object
  This object performs single moves of an ant, which is moving in the graph according to the state transition rule updating the pheromone after the state move.

However, a few changes have been made, and we applied the improved version of the framework on the TSP, a combinatorial problem solved efficiently using ants in [DG97] and [DS04].

3 The Travelling Salesman Problem

In our project we are working with the symmetric TSP, meaning that the distance between two cities a and b will be the same as between b and a. The TSP is known to be a NP-hard problem, so unless we settle for an approximated result, computations will be very time consuming. The easiest way (but as we will see not the quickest way) to find a solution is just to find all the paths, and then choose the shortest one. Unfortunately not many cities are necessary before we end up with an unmanageable number of tours, which again will require an unlimited amount of calculation power. When leaving the first city starting a tour (where the tour will consist of n cities), there will be $n-1$ cities to choose among, and so on after the next city has been visited. This will end up giving us $(n-1)!$. But as back and forth is the same (because of the symmetrical nature of the problem), we can divide it by two and get the expression:

$$\text{Number of tours} = \frac{(n-1)!}{2} \tag{3.1}$$

This quickly gives a lot of tours[2] as it is illustrated in table 3.1 below.

n	Number of different tours
3	1
5	12
7	360
10	181440
15	43589145600
20	60822550204416000
25	310224200866620000000000
30	4420880996869850000000000000000
35	147616399519802000000000000000000000000
40	10198941040598700000000000000000000000000000000
45	1329135787394220000000000000000000000000000000000000000
50	304140932017134000000000000000000000000000000000000000000000000

Table 3.1: Number of nodes vs. number of tours

Such a method, which will end up giving the optimal solution, is obviously not very feasible because of the time consumption required to calculate all the tours. If we assume that we can calculate one tour per nanosecond (10^{-9} seconds), the time consumption will be as seen in table 3.2.

[2] As comparison it can be mentioned that there are app. 4E+79 hydrogen atoms in the universe (http://www.madsci.org/posts/archives/oct98/905633072.As.r.html)

n	Tours	Time in years
20	6,08E+16	≈ 2
25	3,1E+23	≈ 9,84E+6
30	4,4E+30	≈ 1,4E+14
35	1,48E+38	≈ 4,68E+21
40	1,02E+46	≈ 3,23E+29

Table 3.2: Time required to calculate tours if one tour requires 1 nanosecond

It can be seen that even for small instances, the time consumption is extremely high if we want to find every possible tour. Instead we can use an approximation algorithm, which in less time will end up giving a result that isn't necessarily the best tour, but instead a tour that is close to the best tour.

In the previous chapter, we defined a common structure for ant colony algorithms; we are aiming in this section to describe the use of that structure based on mathematical expressions given in [DS04] for solving the TSP. An important element that has to be followed during the implementation of the tour construction is a heuristic list. It defines the constraint satisfaction described in the previous chapter which is, for the TSP, not visiting a node twice. Such constraint helps visiting all the nodes before returning to the initial node.

3.1 A heuristic tour construction

The TSP is also known as a hamiltonian cycle[3] where the problem is to find the minimal spanning tree[4]. An approximate algorithm for solving the TSP is a heuristic construction based on the following:

1. Compute the cost for traversing an edge.
2. Select the minimum cost for a set of edges in the graph that form a cycle

The nearest neighbor is a simple heuristic construction where starting from a random city, the next city visited is the closest unvisited city until the last unvisited city. At the last unvisited city, the next move is to the starting city. The time complexity of the nearest neighbor is based on the visit of all the nodes n and their neighbors $n-1$, giving a quadratic computation time of $O(n^2)$. There are other several heuristic algorithms that are more efficient than the nearest neighbor with better computation time such as:

- The greedy algorithm, where the solution is constructed from the set of sorted edges
- The christofides algorithm with solutions constructed by an Eulerian cycle[5], where nodes are not visited twice.

An ant colony algorithm is another heuristic method for solving the TSP, that has also been proven in [DG97] to be efficient for better tour construction.

[3]This is a cycle where each node of a graph is visited exactly once
[4]This is a minimal set of edges connecting all the nodes
[5]An Eulerian cycle is a cycle where each edge of a graph is visited exactly once

The use of the ants is to construct a heuristic solution following a number of iterations.

3.2 Heuristic Search List

An ant colony algorithm uses two heuristic lists for instantiating the moves of ants in the graph. The two lists are the neighborhood list and the heuristic choice list, also known as the tabu list.

3.2.1 Neighborhood List

A heuristic tour construction starts from a tour which is getting improved as long as the algorithm runs. The neighborhood list is used to improve the minimal tour construction starting from the nearest neighbor tour, as we believe that the overall best tour may contain a subset of the nearest neighbor tour. Moreover, the neighborhood list is used to determine the length of the nearest neighbor tour used for setting the initial deposit of the pheromone by an ant on an edge in the local update rule (see equation 3.5).

The same list can also be used for further optimizations such as 2-opt and 2,5-opt (see chapter 4 for details).

3.2.2 Choice List

The choice list is the list of cities the ant has to visit to perform a tour. The characteristic of the choice list is that once an ant has chosen the next city to visit, that city is removed from the choice list. The procedure of searching in the list ends when there are no more cities to visit; at this point the ant has to go back to the starting city. In Chirico's implementation of the TSP, there were no consideration of the return of the ant which led to unrealistic results.

The above mentioned lists are relevant for the state transition rule, as it is the decision rule of an ant for the choice of the next unvisited city.

3.3 The State Transition Rule

In the TSP state transition rule, the move to the next state is determined by a value q randomly distributed over an interval [0,1] and another value Q_0 also set between [0,1]. The two values are compared, and their comparison leads to one of the two possible rules:

1. Exploitation if $q \leq Q_0$.
   The exploitation is the maximum value obtained by combining the concentration of the pheromone on an edge with its heuristic desirability.

2. Exploration if $q > Q_0$.
   In this rule, the transition is based on the choice of the city with the highest probability using the probability expression or the random proportional choice defined in equation 3.3.

The following expression denotes the description given above of the nodes' transition states for the TSP.

$$j = \begin{cases} argmax_{u \epsilon J_i^k} \{[\tau_{iu}(t)] \cdot [\eta_{iu}]^\beta\} & \text{if } q \leq Q^0 \\ J & \text{if } q > Q_0 \end{cases} \tag{3.2}$$

$\eta_{i,u}$ is the inverted length between the nodes i and u.
τ_{iu}, the amount of pheromone on the edge (i, u).
t, the iteration number.
J_i^k, the set of cities to visit by ant k at city i or the choice list.
$u \in J_i^k$, a city randomly selected.
J, the most probable node to be chosen by an ant while being at position i.
α and β are used for tuning the expression in 3.2 and 3.3. According to [DS04], for $\alpha = 1$ and $\beta = 2$, the tour constructed is similar to the greedy construction defined above. In fact, by setting β to 2 and α to 1, only small distances will have a high concentration of pheromone and by using a choice list, only edges with lower concentration will remain.

$$P_{i,J}^k(t) = \frac{[\tau_{i,J}(t)]^\alpha \cdot [\eta_{i,J}]^\beta}{\sum u \in J_i^k [\tau_{i,u}(t)]^\alpha \cdot [\eta_{i,u}]^\beta]} \qquad \text{if } J \in J_i^k \tag{3.3}$$

3.4 Pheromone Update rule

By distinguishing the local update as an update made by an ant to improve path search, and the global update as a reinforcement of an iteration's best tour, the following update rules are made for the TSP:

3.4.1 The local updating rule

For the TSP, Chirico expressed the parameter of deposit by combining the sum of all the average lengths between the nodes and the number of nodes to obtain the following:

$$\tau_0 = \frac{2 \cdot n}{\sum \delta(r, s)} \tag{3.4}$$

n is the number of nodes.
$\delta(r, s)$, the length between node r and node s.
Chirico did not expand on why he had used this expression, so we decided instead to chose the expression 3.5 given in [DG97] and [DS04], where the parameter of deposit is based on the length constructed by the nearest neighbor list and the number of nodes.

$$\tau_0 = (n.L_{nn})^{-1} \tag{3.5}$$

With n as the number of cities and L_{nn} the length produced by the nearest neighbor list.
Equation 3.5 is used to spread the pheromone using equation 3.6 on each edge used by an ant. The fact that the pheromone is spread along all edges traversed by an ant opens for all possible solutions.

$$\tau_{i,j}(t) \leftarrow (1 - \xi) \cdot \tau_{i,j}(t) + \xi\tau_0 \tag{3.6}$$

ρ is the parameter of decay and t is the iteration number when the update is performed.

3.4.2 The global updating rule

We set the global update to be:

$$\tau_{i,j}(t) \leftarrow (1 - \rho) \cdot \tau_{i,j}(t) + \rho\Delta\tau_{i,j}(t) \tag{3.7}$$

where ρ is still the parameter for the pheromone decay and $\Delta\tau_{i,j}^k$ a parameter of deposit defined by:

$$\Delta\tau_{i,j}^k = \frac{W}{L_{best}} \tag{3.8}$$

Where L_{best} is the length of the best cycle, and W is a parameter ranged between [1..100]. Our choice of W is based on the settings of Bonabeau et al. described in [JM03].

3.5 TSPLIB

The tests made using the TSP algorithm are based on real cases using TSPLIB as a reference. TSPLIB is an online library, developed by the university of Heidelberg in Germany that contains several samples of TSP and similar related problems ranged on a list of different files. It has become a standard reference in modern research and the documentation can be obtained in [Rei95]. We chose solely to focus on instances of the type Symmetric Euclidian TSP, referenced as EUC_2D, meaning that distances (or weights) between nodes are expressed on an Euclidean 2D coordinate system. The coordinates are decimal numbers (or doubles), including negative values and the distances between the nodes are computed according to the Pythagoras equation.

Based on the framework structure, we extended the TSP for tour optimizations and implemented methods for local search based on the 2-opt and 2,5-opt algorithms.

4 Local search optimization

No matter how effecient a TSP algorithm is, it will in theory always have some shortcomings, as the updating rules can not possibly take all situations into account when being designed. Therefore it will always be an advantage to set an optimization method, which can be used after retrieving an initial result from the TSP algorithm for improvement. The heuristic used for optimizing the TSP is local search.

The most commonly used optimization algorithms for the TSP are 2-opt, 2,5-opt, 3-opt and the Lin-Kernighan. The Lin-Kernighan algorithm is generally seen as being the most efficient optimization algorithm right now, particularly after Keld Helsgaun created and published his own implementation[Hel98]. Its efficiency is to be seen in its complexity, which convinced us from the beginning, that even if it could have been better to get it involved in the project, we would have to drop it as we didn't have the required time and experience to implement it. Instead we turned ourselves towards implementating a 2-opt, hoping to have time to also implement a 2,5-opt and 3-opt algorithm. As time went by, and despite the simplicity of the 2-opt algorithm and implementation, we ended up spending a lot of time struggling to get it to work properly and optimizing the code. This resulted in that we after this only had time to implement a 2,5-opt algorithm, meaning that we had to give up on trying to implement a 3-opt algorithm.

4.1 2-opt

The 2-opt is a basic case of the local search optimization heuristics, and as such it is capable of obtaining useful results very fast. Even if the other more advanced options, such as the Lin-Kernighan, would give us a higher chance of a near to optimal (if not the optimal) tour, the 2-opt is a feasible choice when looking at its results versus the time required to obtain these results. The implementation of the 2-opt is based on the following points as shown in figure 4.1:

1. Take two pairs of consecutive nodes, pairs A & B and C & D from a tour.
2. Check to see if the distance $AB + CD$ is higher than $AC + DB$.
3. If that is the case, swap A and C, resulting in reversing the tour between the two nodes.

The tour should be run through from the beginning to check for any possible swaps every time a swap is made, as every swap results in a new tour being made. The swap can be performed in two different ways:

- Search until the first possible improvement is found, and perform the swap.

- Search through the entire tour to find all possible improvements, and perform only a swap on the best improvement.

We chose to use the first option (as seen in code fragment 4.1), as the second one could possibly run for a much longer time before returning a result, as it has to run through its entire list of neighbors before it performs a swap, whereas the first on performs the swap as soon it hits the first possible improvement. The disadvantage of choosing the first one over the second one is that we might loose a potential good improvement.

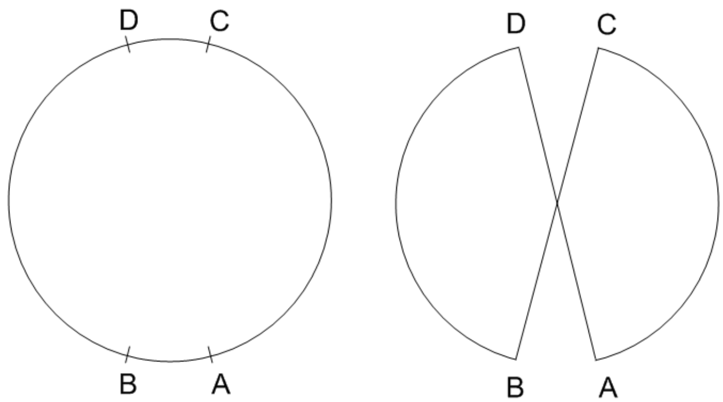


Figure 4.1: A 2-opt operation.

For shorter tours it is feasible to let the algorithm run until it cannot find another swap, but for larger tours it is recommended to implement a check which at some point during optimizing should go in and stop the process, as it would run for an undesirably long time. This could be a simple limit on how many swaps should be done until it stops. The disadvantage of doing it like this is that the chances for achieving the optimal tour length decreases dramatically, as you are not letting the program run undisturbed until it can't find more possible optimizations. On the other hand you limit the runtime and are therefore not forced to wait for an unknown amount of time before it completes the run.

As it can be seen the codes themselves are simple, and it is very easy to recognize what is going on. As mentioned earlier, the complexity of the code itself is not what requires the long computation time, but rather the actual calculations and operations that have to be done as the algorithm works itself through the tour.

According to [DS04, page 94], the time complexity for running a neighborhood search in 2-opt is $O(n^2)$, which is significantly lower than the 3-opt's $O(n^3)$. Using various optimization techniques these times can be lowered, but as the algorithm says above, a neighborhood search will have to be performed for every node in the tour. The version we have implemented has already been optimized in respect to how the algorithm originally was conceived, as using the nearest neighbor list is part of the optimization techniques mentioned in [Nil03]. By doing that it is possible to limit how many nodes each node should check when looking for a possible improvement. We chose from the beginning to use the complete neighbor list to make sure we don't miss a possible swap, but in doing that we have not saved any time compared to if we hadn't implemented the

Code 4.1: The 2-opt algorithm

```
do {
    node A = first node (any node) in the tour
    do {
        node B = A.next
        for (each of B's neighbors) {
            node C = B's neighbor
            node D = C.previous
            if ((distance(A,B)+distance(C,D)) >
                (distance(A,D)+distance(B,C)) {
                swap()
                break so the algorithm starts over again
            }
        }
        A = A.next
    } while (A != first node)
} while (there has been made changes)
```

Code 4.2: The 2-opt's swap algorithm

```
node temp
for (all the nodes from A to C){
    temp = node.next
    node.next = node.previous
    node.previous = temp
}
```

neighbor list. Instead we could have limited the list to only contain 20% of the total number of nodes, decreasing the required computation time greatly, but also increasing the risk that we won't end up having a fully optimized tour. According to [JM97, page 26] the improvemnent in a tour when going from 20 to 80 neighbors is only app. 0,1-0,2% on average, which means that our concerns about not finding all the possible improvements were unnecessary. This changes the time complexity for the 2-opt from $O(n^2)$ to $O(nm)$, where m is the number of neighbors.

4.2 2,5-opt

The concept of the 2,5-opt algorithm is simpler compared to the 2-opt algorithmas as it only performs a move of a single node. The simplicity of the algorithm affects the results that can be gained by using the 2,5-opt, which is why our opinion is not to use this optimization algorithm alone, but rather use it in combination with another, which in our case is the 2-opt. But it can be useful as a finishing touch; after running another optimization algorithm, it can be used to find small improvements throughout the tour that the former optimization did not find, thus decreasing the distance a little more. The structure of the 2,5-opt algorithm is as seen in figure 4.2:

1. Take two consecutive nodes A and B.
2. Check to see if the distance is decreased if C is moved in between A and B.
3. If that is the case, insert C in between A and B.

From the visual representation in figure 4.2, it is obvious that the 2,5-opt only performs a simple move of a node, solely dependent on that the distance $AB + CD + DE$ is higher than the distance $CE + AD + BD$.

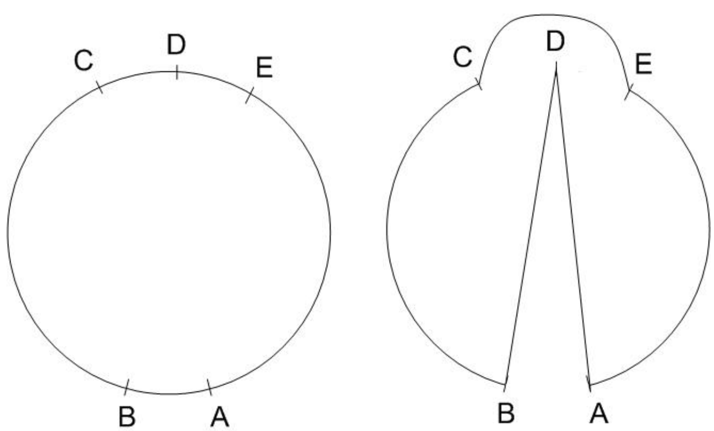


Figure 4.2: A 2,5-opt operation.

Its simplicity can be seen in the code fragment 4.3.

Code 4.3: The 2,5-opt algorithm

```
do {
    node A = first node (any node) in the tour
    do {
        node B = A.next
        for (each of A's neighbors) {
            node D = B's neighbor
            node C = D.previous
            node E = D.next
            if ((distance(A,B)+distance(C,D)+distance(D,E)) >
                (distance(A,D)+distance(B,D)+distance(C,E)) {
                swap()
                break so the algorithm starts over again
            }
        }
        A = A.next
    } while (A != first node)
} while (there has been made changes)
```

Code 4.4: The 2,5-opt's swap algorithm

```
C.next = E
E.previous = C
A.next = D
B.previous = D
D.previous = A
D.next = B
```

Even if the actions done by the algorithm are simpler than those of the 2-opt algorithm, the complexity of the code is slightly higher, as another node is to be involved to enable the computations. But this does not change the fact that the algorithm only affects these 5 nodes, whereas the 2-opt algorithm impacts not only the 4 named nodes, but also all those nodes between A to C.

Comparing to the potential time consumption of the 2-opt algorithm, the 2,5-opt therefore has the advantage that less computation time is needed for changing nodes, as only 5 nodes are to be changed. A way to optimize the 2,5-opt can also be done by using a neighborhood search similar to the one in the 2-opt, with the results in a time complexity of $O(n^2)$, or $O(nm)$ if you chose to limit the neighbor list.

5 Implementation

In this chapter we will introduce the functionality of Chirico's original program, followed by introducing our version with the changes we have made including extra classes we implemented to get the functionality we wanted.

5.1 The original code

The original code has an implementation of an ACS framework, including solutions for the TSP and SP[6] that take advantage of the ACS. Since we didn't investigate the SP, we will only focus on the ACS framework and the TSP. Figure 5.1 gives an UML diagram of the framework structure and its TSP extension.

The program is run through a command prompt where the required inputs are the number of ants, nodes, iterations and repetitions. Before creating a `Graph` object and starting the `AntColony`, the number of nodes is used to create a `delta` matrix which is the matrix defining the distance between the nodes. These distances were calculated based on a random number generator, so the results retrieved from the program were not comparable in any way with the official instances found on TSPLIB.

5.1.1 The ACS framework

1. The graph
   When starting the application, an object of the type `AntGraph` is created, containing the matrices `delta` described above and `tau` for setting the pheromone on the edges of the graph. The class also contains methods that enables changes in these matrices during runtime, such as updating the pheromone on the edges and resetting `tau` for a new repetition.

2. The ant colony
   After creating the graph, an `AntColony` object is created, with the graph as one of its parameters. This way, the ants can - through the colony - always get access to the graph so they know what options they have when going to their next node. The colony keeps track of all the ants associated with this colony (as the framework supports more than one colony at a time), which is accomplished by having an array of `ant` objects. The ant colony also stores information of the best tour performed by the ants at each iteration. Before the first iteration is run, the abstract method `createAnts` is called

[6] The Steiner Problem is similar to TSP, but opposite TSP where you only have the nodes you are supplied from the beginning, the SP allows for creating temporary nodes, that can shorten the tour between several nodes. Eg. would a tour between three nodes using the SP be shorter than when using the TSP, as you can put a node in the middle, letting the edges connect via this. This way all the nodes are connected, but non of them are directly.

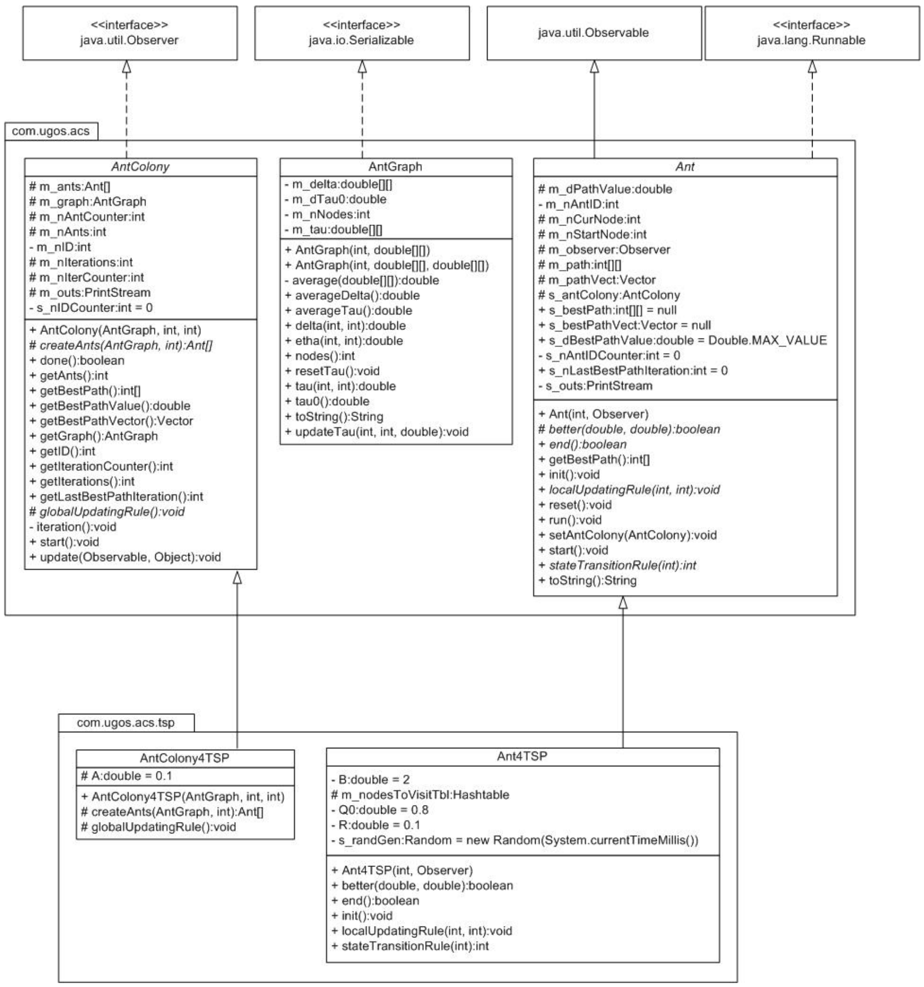


Figure 5.1: UML diagram of the old version of the program.

to create the ants, taking the graph and number of ants to be created as parameters. All the iterations are then run through, and each iteration begins by starting all the ants, which report back to the colony using the `update` method when they have created a new tour. When the ants have been started the abstract method `globalUpdatingRule` is called, applying the global updating rule to the graph's edges. When all the ants have reported back and the global updating rule has been applied, a new iteration is started if there are still any left according to the colony's iteration counter.

3. The ant
   The `Ant` object holds information about the start node, the current node, the tour list[7], its best tour so far and the iteration the best tour was made. When the `start` method for an ant is called, a new thread is created with

[7]The tour list is the list containing the nodes the ant has visited so far during the iteration.

the ant as parameter. The thread is then run, finding the next node for the ant using the abstract method `stateTransitionRule`. At the end of the method call, it calculates its new tour length based on the distance to the current node, adds the new node to its tour list, and deposits pheromone on that edge according to the abstract method `localUpdatingRule`. At the end of the iteration it checks to see if the new tour is shorter (abstract method `better`) than the best tour so far, and if this is the case, the new tour becomes the best tour so far.

5.1.2 The TSP implementation

The TSP algorithm for the original code is fairly simplified, as the basic functionality of the ant already exists in the framework. As an extension to the framework, it is only supplying rules for the ants' behaviour.

1. The ant colony for TSP
   As described in figure 5.1, the `AntColony4TSP` class only holds its constructor and implements the two abstract methods `globalUpdatingRule` and `createAnts`. The colony creates the ants of the type `Ant4TSP`, resets the ants (by resetting the values for the ants' best tour), associates them with this colony, and sets the starting node as a random one. The global updating rule algorithm used for this TSP solution can be seen in [Chi04, eq 4].

2. The ant for TSP
   The `Ant4TSP` is an extension of `Ant` that overrides the `ant`'s initialization or the `init` method, adding a `Hashtable` of nodes the ant needs to visit, and sets an `end` condition to true if the table is empty. The `localUpdatingRule` and `stateTransitionRule` methods are performed following the rules described in [Chi04]. The implemented `better` method simply compares two final tour distances and select the best.

5.2 Our changes to the code

We very soon found limitations in the original framework; because the nodes were represented by integers, they would not be able to carry any data, which would make it impossible to give them any coordinates. This quickly convinced us that we had to change how the framework handled the nodes, as we would replace the integers with `Node` objects instead. As we were dealing with TSP files, we also set the framework to be compatible with the files found on TSPLIB. For a view of the tour, we implemented a GUI and added buttons for setting parameters without having to go into the Java files. A final goal was to implement one or more local search optimization algorithms in an attempt to give us better chances of getting an optimal solution. We ended up with a structure described by the figures 5.2, 5.3 and 5.4. We will be going through the framework and TSP algorithm stating only the changes that have been made. We also applied changes to the class names, as we were warned about a possible mis-interpretation for an object-oriented framework[8].

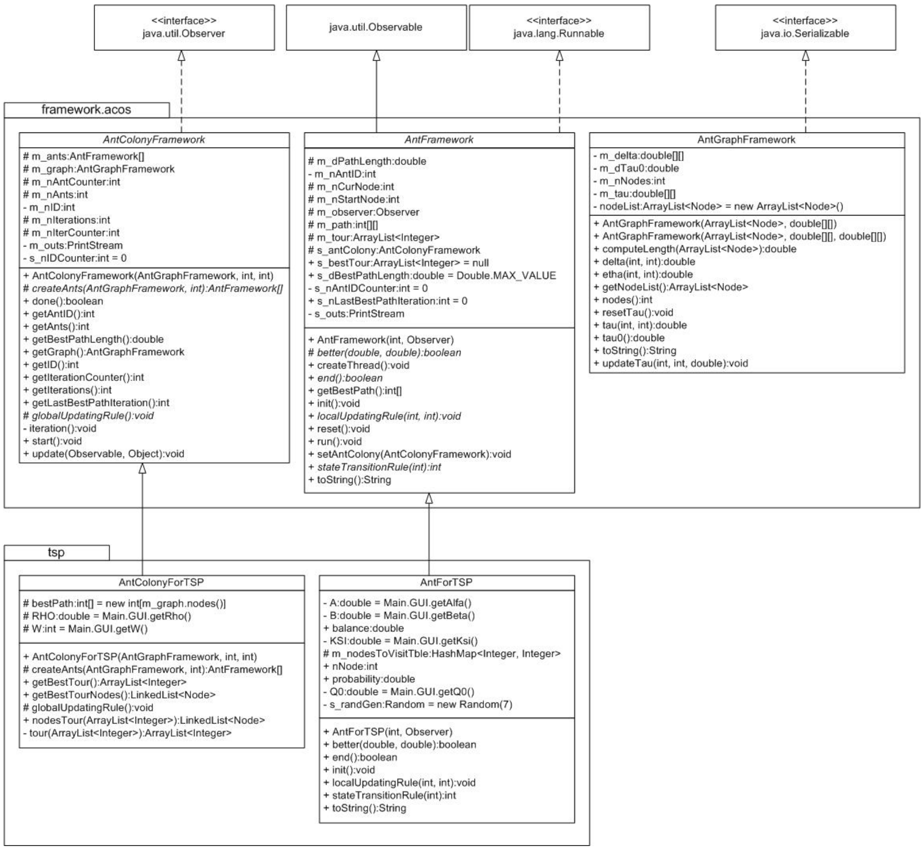


Figure 5.2: UML diagram of the new version; the framework.acos and tsp packages.

5.2.1 The ACS framework

1. The graph class
   To have the list of nodes with their coordinates and ID available at all times in the program, we chose to change the constructor of the `AntGraphFramework`, so instead of taking the number of nodes in the instance, it takes an `ArrayList` containing all the `Node` objects. Another change made was the way the framework computes τ_0 as described in chapter 3. We implemented a method called `computeLength` for finding the length built by the nearest neighbor list as seen in the code (see 5.1).

2. The ant class
   The `AntFramework` class is almost identical to the `Ant` class. Some minor changes were made such as the replacement of the `Vector` type to an `ArrayList` to easily manage the `Node` object. We renamed the `start`

[8] According to our supervisor Keld Helsgaun.

Code 5.1: `computeLength` method

```
while (tabuList.size() != 1) {
    currNode.setNeighbors(myNodes);
    for (Neighbor nabo : currNode.neighbors) {
        if (tabuList.containsKey(nabo.toNode.nodeID)) {
            length += currNode.distance(nabo.toNode);
            tabuList.remove(new Integer(currNodeID));
            currNode = nabo.toNode;
            currNodeID = nabo.toNode.nodeID;
            break;
        }
    }
}
length += currNode.distance(firstNode);
return length;
```

method to `createThread` to avoid confusion with the threads' start method, and we removed the `int[][] s_bestPath` in the global updating rule, since we ended up not using it.

3. The ant colony class
   Also the `AntColonyFramework` is very much alike its predecessor `AntColony` with only a very few changes done to it. We added an access point to the ant counter from outside the class, removed access to the tour vector because the tour could be retrieved directly from the ant object instead, and we also removed the acces to the path array, `int[][] s_bestPath`, for the reasons described above.

5.2.2 The TSP algorithm

Our TSP implementation has gone through some more extensive changes than the framework, as a result of actually simulating the basic behaviours of the ants. Apart from the algorithms, the most noteworthy changes have been made to accomodate the `Node` objects instead of `Integer`s, and making the parameters in the algorithms get their values from the GUI.

The local updating rule of the `AntForTSP` class has not been changed if looking at the computations that are done, but the ρ has been swapped with ξ to follow Dorigo's terminology as seen on page 78 in [DS04]. The only big change as such has been done in the state transition rule method. First of all it now checks if the current ant is the first ant in the first iteration; if that is the case it will be doing a tour using the nearest neigbor heuristic. Otherwise the ant has the choice between exploitation and exploration; the former is unchanged compared to the one in `Ant4TSP`, but the latter has been changed. Instead of using the equations found in [Chi04, eq 1&2], we implemented the equation 3.6 as seen in code fragment 5.2.

The `AntColonyForTSP` class has had some extra functionality added which provides retrieval of the best tour in a list, either filled with `Integer`s being node

Code 5.2: The local updating rule

```
public void localUpdatingRule(int nCurNode, int nNextNode) {
    final AntGraphFramework graph = s_antColony.getGraph();
    double val = ((double) 1 - KSI) * graph.tau(nCurNode,
                        nNextNode) + (KSI * (graph.tau0()));
    graph.updateTau(nCurNode, nNextNode, val);
}
```

Code 5.3: The global updating rule

```
protected void globalUpdatingRule() {
    double dEvaporation;
    double dDeposition;
    for (int i = 0; i < m_graph.nodes(); i ++)
        bestPath[i] = AntForTSP.getBestPath()[i];
    for (int r = 0; r < m_graph.nodes(); r++) {
        for (int s = r + 1; s < m_graph.nodes(); s++) {
            for (int i = 0; i < super.getAnts(); i ++) {
                double deltaTau =
                        (W / AntForTSP.s_dBestPathLength);
                dEvaporation = ((double) 1 - RHO) *
                        m_graph.tau(bestPath[r], bestPath[s]);
                dDeposition = RHO * deltaTau;
                m_graph.updateTau(bestPath[r], bestPath[s],
                        dEvaporation + dDeposition);
            }
        }
    }
}
```

IDs, or actual `Node` objects. The global updating rule method has been changed slightly; $\Delta\tau$ is now calculated as seen in 3.8. The implementation of the evaporation and deposition have also been changed in accordance to equation 3.7 and code fraction 5.3.

5.2.3 Additions to the code

We put the additions to the code into 4 packages; Node, IO, tsp.optimization and Main.

The `Node` package

The `Node` package contains two classes; the `Node` and `Neighbor` classes. The node class creates a node object which contains the node's coordinates and ID. It also has the possibility of assigning a list of `Neighbor` objects sorted by the distance to the node. The neighbor is just a node object put together with a distance to the node to which the neighbor belongs.

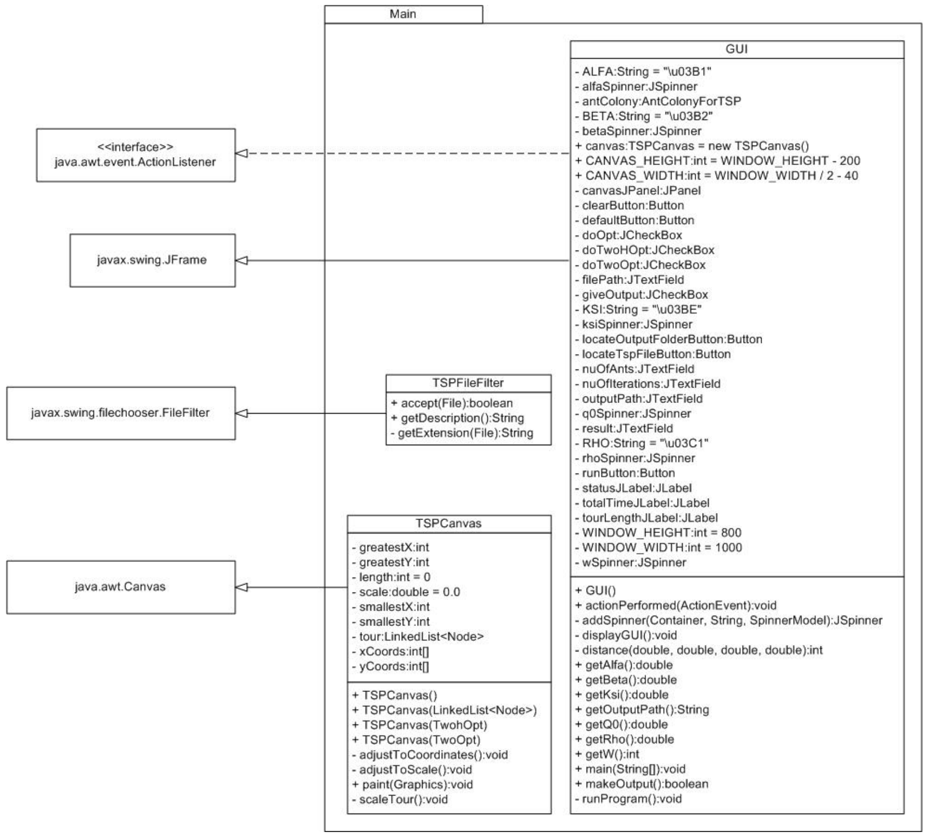


Figure 5.3: UML diagram of the new version; the Main package.

The `IO` package

The program accesses the external tsp files via the `InputFile` class in the `IO` package. This class takes the path and filename of a file, takes the data found in the file and converts it into `Node` objects and puts them into an `ArrayList` which can be retrieved using the `getNode` method.

The `tsp.optimization` package

Any optimization of a tour is done in the `tsp.optimization` package, where the classes `TwoOpt` and `TwohOpt` resides. These perform a 2-opt and 2,5-opt optimization on a given tour, respectively. For more details on how the optimizations are performed, please refer to chapter 4 on page 15. For the `twoOpt` our method ended up looking like the pseudo code 5.4.

As it can be seen, the algorithm is not as efficient as it can be, as we are handling a lot of lists - lists that we could have avoided had we had the skills and time to implement a more effective algorithm. The swap for the 2,5-opt on the other hand was very simple, as we were using a `LinkedList` to hold the tour:

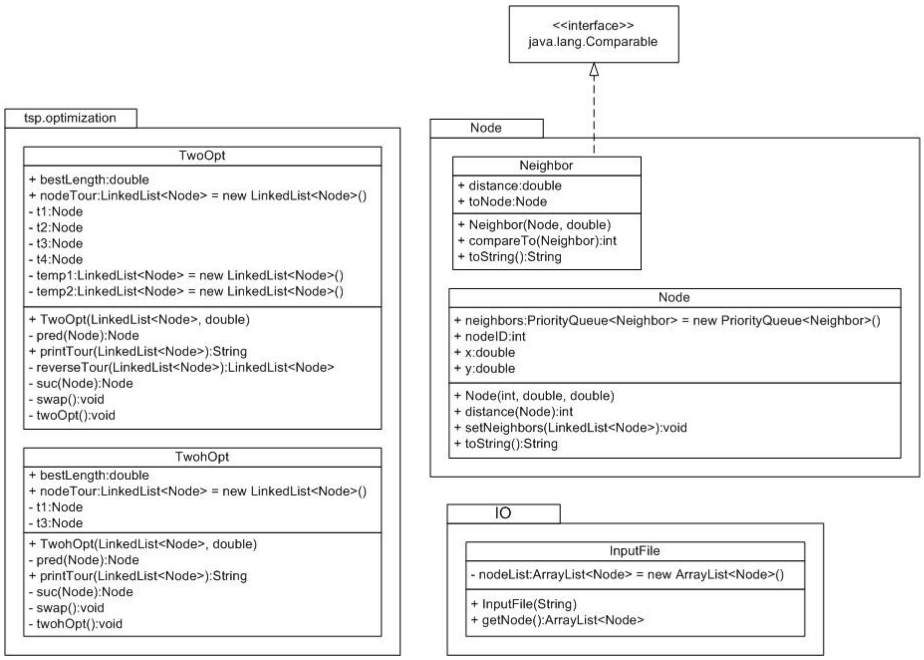


Figure 5.4: UML diagram of the new version; the Node, IO and tsp.optimization packages.

Code 5.4: Our swap method for the 2-opt

```
ArrayList temp1, temp2;
node = T2
do {
    insert node into temp1
    node = node.next
}
while (node != T4}
insert T4 into temp1
node = T3
do {
    insert node into temp2
    node = node.next
}
while (node != T1)
empty the tour list
insert temp2 and reversed temp1 into the tour list
```

Having the add and remove methods already built in the `LinkedList` object, we are saved from a lot of work of changing the properties of T3's neighboring nodes.

Code 5.5: Our swap method for the 2,5-opt.

```
remove T3 from the tour
add it to the index after T1
```

The `Main` package

The `Main` package contains the `GUI`, `TSPCanvas` and `TSPFileFilter` classes, which exist to make it easy for the user to control the program, as he in a user interface can find the TSP instance file he wants to use, define the parameters ρ, α, β, ξ, $Q0$ and W, choose the number of ants and iterations, choose what kind of optimization he wants to have done on the tour (if any), and lastly get a graphical representation of the tour when it has been found together with its length.

The `GUI` is the main classs containing means for managing the values of the parameters. The `GUI` extends `JFrame` which utilizes action listeners for tracing user interactions with the GUI. The inner class `InputVerifier` makes sure that the user only enters valid integers when choosing the number of ants and iterations. The `TSPFileFilter` class makes the file chooser (which is used when looking for a tsp file) only show folders and tsp files. The `TSPCanvas` is our customized version of `Canvas`; it takes a tour, `twoOpt` or `twohOpt` object, and draws the optimal tour on the canvas, scaling it so that it fits in the canvas, whithout getting stretched.

5.3 Known bugs

Even if we have been spending a lot of time to get the code working the way we wanted, we unfortunately still experience problems we haven't been able to or had time to solve before we got too close to the deadline. The known problems are:

- The scrollbar in the GUI.
  We have for some reason not been able to get the scrollbar to work properly so that the user could scroll back and forth in the list of nodes to see in what order they come in the final tour. This problem can be circumvented by selecting the text and dragging the selection to the left or right (or just pressing home/end). This is of course not a preferable way of doing it, but until we figure out how to use the `ScrollBar` and `ScrollPane` in the Java library, we have no solution for it at this time.

- dSum = Infinity
  Occasionally when running the program, we experienced getting an `ArrayIndexOutOfBoundsException: -1` when the ant was to retrieve its next node from the state transition rule method. We found out that at random, `dSum` would be set to infinity because the distance between `nCurNode` and `nNode` was 0. After some more investigation we found that when the error occured, the `m_nodesToVisitTble` still contained the `nCurNode` even

though it should have been removed when it was the ant's next node. It happened when there was only one node left in the `m_nodesToVisitTble`, as there wouldn't be other nodes to be checked to see if they were above 0 (the `balance`). So as long there were more than one node left in the `m_nodesToVisitTble`, this bug would not have an affect on the overall functionality, whereas when there was only one left, it stopped the program.

To avoid this there should have been an extra check when deciding whether the ant should explore or exploit; if the `dSum` was Infinity, the ant should be forced to exploit, completely avoiding the calculations in which it could do damage.

It is unknown at this time what actually triggers the program not to remove a visited node. We have conducted several tests printing out data on the ant, but there seems to be no common thread between the different occurences of the bug.

6 Experiments

After modifying Chirico's framework and TSP implementation, we wanted to investigate how well our program would perform compared to other existing solutions. From the beginning we had chosen to take Chirico's original application, as it would be interesting to see if we in any way had improved its performance, or just added extra functionality[9]. As another reference, we took a solution called SimpleACS [BDDW02], as it looked interesting because it was obvious that its goal is not to be a framework, but rather just an ACS solution without any thought of making it easy for other people to expand it at a later time.

6.1 Test setup

The tests were run on a Pentium 4 3,0 GHz with 512 MB RAM and Windows XP with all the latest updates installed. The JRE used was Sun's own version, and the version used was the newest at the time of testing; version JDK 1.6.0_01-b06. All times are in seconds unless otherwise stated.

For the parameters (ξ, ρ, α, β, W and $Q0$) we tried to find the perfect combination to get a result that was as optimized as possible, but as one could imagine, this would require an extensive amount of testing time and patience, something we did not have either of at the time. Instead we chose to apply the following parameters as suggested in [DS04]:

$\rho = 0.1$ $\alpha = 0.1$ $\beta = 2$ $Q0 = 0.9$

And ξ was set to $0,9$ as suggested in [DG97], as they claim to have had best experience with these. The value for W being set to **??**1 came from the fact that it replaced the value 1 (or more accurately being multiplied with $\frac{1}{C^{bs}}$ as seen in [DS04, page 74]), so keeping it as 1 is a result of unfortunate lack of testing.

The instances used for the tests were all found on TSPLIB, and supplied on the enclosed CD.

6.2 Tests

When wanting to test our program to see how it performed, we chose to only use one case for testing, as we expected that changing the number of ants or iterations would affect the outcome equally for all the cases. We chose to use the eil51 case as our test subject for several reasons. Firstly we have been using this case (or parts of it) througout the entire development period for testing our

[9]Chirico's unedited code can be seen in appendix D and also found on the enclosed CD

algorithms, so we knew what kind of results to expect, and secondly it has a size that makes it easy to test because it won't take too long to perform a batch of tests, but it is on the other hand no too small either.

We did all the tests using 10 ants and 10 iterations, and unless otherwise stated we also applied optimization to the tour.

We wanted to see how big deviations we would get when running the program several times both with and without optimization. As seen in figure 6.1 we get an average of app. 433 when using optimization, which is 7 higher (making it a deviation of 1,73%) than the optimal of 426. Our results vary from 427 (0,23% deviation) to 443 (3,99%), and considering more than half of the runs are below average, the distribution of results is fairly good.

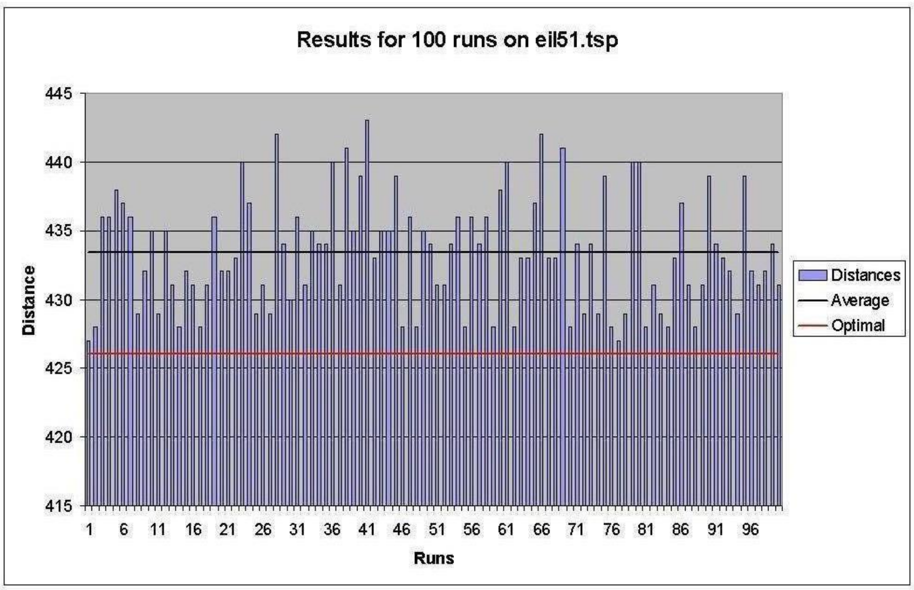


Figure 6.1: Running the eil51 instance 100 times using optimization.

When we don't use our options to optimize the tour, the results change dramatically, and the results are far from as positive. As seen in figure 6.2 the results vary between 462 and 509, with an average of app. 483 (8,45%, 19,6% and 13,44% deviation respectively). Obviously the extra time required to do the optimizations on the tour made by the TSP algorithm is well spent when looking at the gained improvement.

We wanted to see how changing the number of ants and iterations would affect the time the program would need to complete a task, and also if it would affect the output distance. It could be interesting to see if it would be worth the time to use 100 iterations instead of 10, or how it would affect the resulting distance, and also if the changes were done to the ants instead. This resulted in four tests, where eil51 was run 300 times, and the number of ants or iterations were increased by one for each run, thereby testing it with 1-300 ants or iterations. When the iterations were increased we were using 10 ants and vice versa. The results can be seen in figures 6.3 - 6.6.

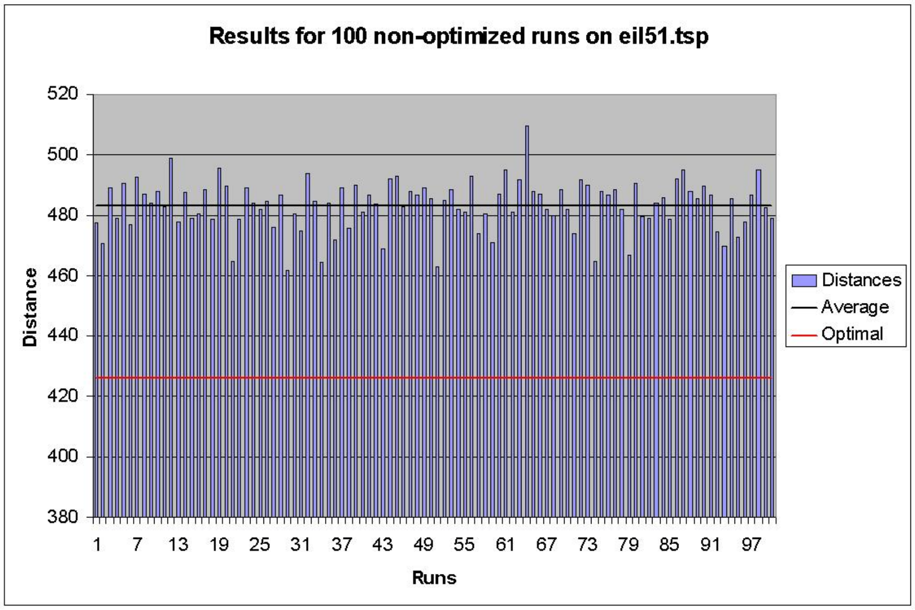


Figure 6.2: Running the eil51 instance 100 times without optimzation.

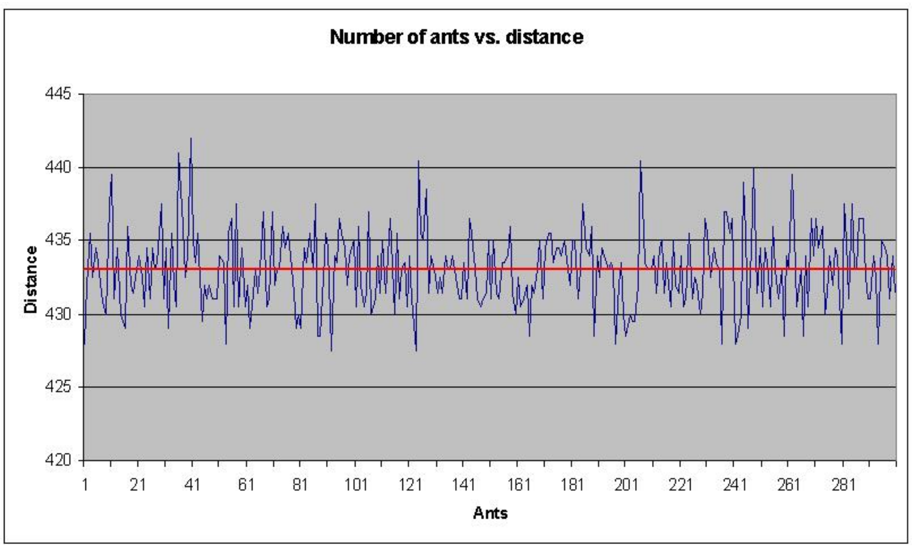


Figure 6.3: Running the eil51 testing the distance with 10 iterations and an increasingly amount of ants.

One would at first think that because we use more iterations or ants, the chances for getting a good result would be higher. Unfortunately that is not what our tests tell us. As seen on figure 6.3 and 6.4 the results seem very random between the two extreme values found in figure 6.1 (427 and 443). It can also be seen that the average value is slightly higher when varying the number of iterations than the number of ants, but we suspect that that might just be because of

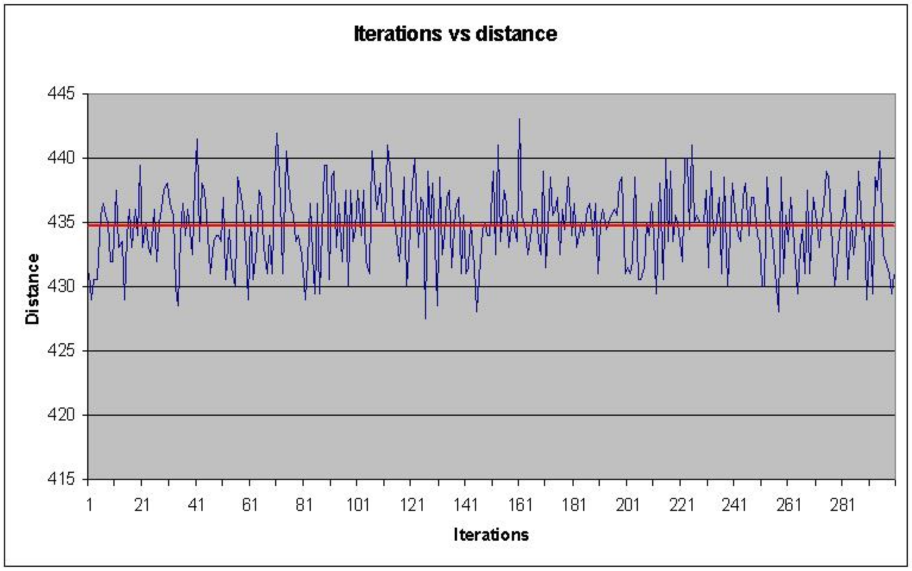


Figure 6.4: Running the eil51 testing the distance with 10 ants and an increasingly amount of iterations.

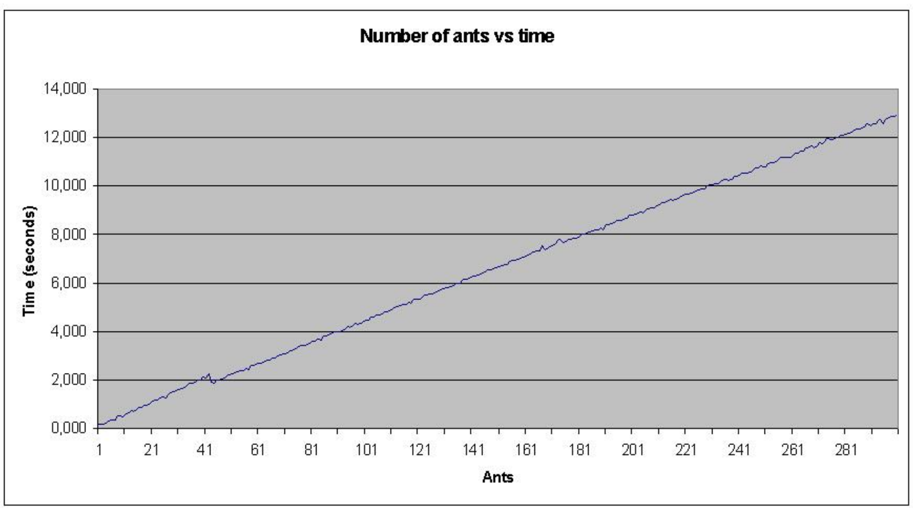


Figure 6.5: Running the eil51 testing the time with 10 iterations and an increasingly amount of ants.

the randomness of the results. We tried also running with 10 ants and 5000 iterations to see what kind of results we would get, and it was clear to us that these results were just as random as those seen in figure 6.3 and 6.4.

Not finding any evident order in the results when changing the number of ants or iterations, we started investigating any relation between the computation time and number of ants or iterations, and we found that it's a completely different case. For both varying number of ants and iterations we got an approximated

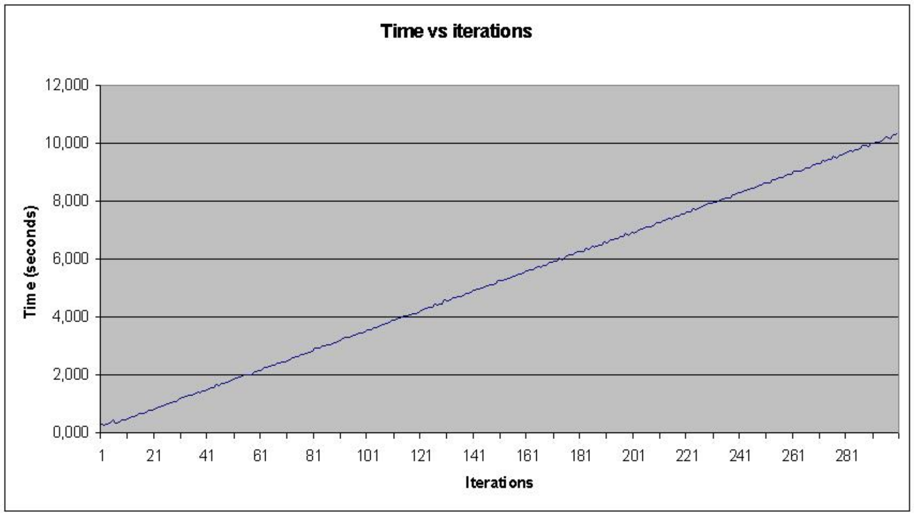


Figure 6.6: Running the eil51 testing the time with 10 ants and an increasingly amount of iterations.

linear ascending line, only with few discrepancies. These can be seen in figure 6.5 and 6.6. Notice that figure 6.5 has a slightly faster ascension than figure 6.6 as a new ant requires more computation power (and therefore more time) than a new iteration.

6.3 Comparison of performance

When comparing our program's performance with the two beforementioned applications, we took all the instances found at TSPLIB of the type Eucledian 2D (which can be found on the enclosed CD), and let the programs try to find a solution. As neither Chirico's program nor SimpleACS implement an optimization algorithm, we deactivated ours so that we would get a more fair result. For the tests we used 30 ants and 300 iterations.

As seen in table 6.1, the SimpleACS surpasses the two other in both time consumption and quality of the output result. The reason for this is that SimpleACS as mentioned is not a generalized framework which needs to be easy to use so it can be extended by other people for other solutions not necessarily thought of by the programmer. Instead it is highly specialized with only one kind of implementation in mind. This heightens its efficiency greatly, and this is reflected in the results seen in table 6.1. Another thing to think of is that SimpleACS is not object oriented, so not having to handle custom objects limits the weight of the program on a computer, giving the program an extra advantage. Compared to Chirico's original program we actually received better results, proving that the changes we made to his program were not only simple changes, but also improvements. On the other hand our program required more time to do the calculations, so in this case one has to decide whether the time spent or the final result is the most important thing.

	Chirico's solution		SimpleACS		Our solution	
	Time	Result	Time	Result	Time	Result
a280.tsp	423,8	3299	14,328	2811	535,609	3108
berlin52.tsp	26,688	8099	0,703	7748	39	8059
bier127.tsp	109,859	136408	3,313	125555	123,125	131570
ch130.tsp	138,437	7586	3,547	6364	133,36	7129
d1655.tsp			478,234	88119		
d198.tsp	308,984	18765	6,968	17337	280,172	16128
d2103.tsp						
d657.tsp	3434,282	74506	84,766	61329		
eil101.tsp	75,062	747	2,172	654	91,859	748
eil51.tsp	26,359	460	0,656	433	28,515	469
eil76.tsp	48,203	603	1,328	547	58,328	615
fl1400.tsp			332,609	24212		
gil262.tsp	368,766	3388	15,047	2640	460,188	2823
kroA100.tsp	75,969	25326	2,156	21585	104,672	25420
kroA150.tsp	160,281	34686	4,641	27602	208	31610
pr1002.tsp			164,375	335872		
pr2392.tsp						
u724.tsp			100,719	55783		
usa13509.tsp						

Table 6.1: Comparison of performance. The empty places are runs where the program encountered a `java.lang.OutOfMemoryError: Java heap space` exception.

We then tried to use our optimization algorithms to see if using them would give us an advantage in both time and quality against the others. This time we only used 10 ants and iterations, as the optimizations would compensate for low quality of the initial result from the TSP algorithm. The results of this test can be seen in table 6.2.

	Without optimization		With optimization		Improvement	
	Time	Result	Time	Result	Time	Result
a280.tsp	535,609	3108	12,578	2675	97,65%	13,93%
berlin52.tsp	39	8059	0,422	7902	98,92%	1,95%
bier127.tsp	123,125	131570	2,125	122440	98,27%	6,94%
ch130.tsp	133,36	7129	2,203	6349	98,35%	10,94%
d198.tsp	280,172	16128	5,797	16017	97,93%	0,69%
d657.tsp			98,89	51678		
eil101.tsp	91,859	748	1,468	652	98,40%	12,83%
eil51.tsp	28,515	469	0,406	428	98,58%	8,74%
eil76.tsp	58,328	615	0,782	543	98,66%	11,71%
gil262.tsp	460,188	2823	9,266	2518	97,99%	10,80%
kroA100.tsp	104,672	25420	1,375	21959	98,69%	13,62%
kroA150.tsp	208	31610	3,11	28098	98,50%	11,11%

Table 6.2: The improvement going from not using to using optimization.

As it can be seen, the improvements in performance were massive in result and especially time consumption. Considering that we save up to 98,96% in time consumption *and* get an improvement of 13,62% in the result in the case of the kroA100 instance, we believe that using optimization algorithms together with basic ACO and TSP solutions can save a lot of time without having to accept a decrease in quality of the tour. It is also interesting to notice, that when running the d657 case not using optimization the program runs out of memory, whereas when we are using our optimization algorithms, the test run completes and gives us a usable result. This is to be viewed in the light of the fact that the amount of ants and iterations when using optimization are down to $\frac{1}{3}$ and $\frac{1}{30}$ respectively compared to when running without, reducing the number of used objects significantly. So not only does the usage of optimization give us better results, it also gives us a better chance of getting a result in the first place.

Instance	Optimal distance	Deviations			
		Chirico's solution	SimpleACS	Our solution	
				without optimization	with optimization
a280.tsp	2579	27,918%	8,996%	20,512%	3,722%
berlin52.tsp	7542	7,385%	2,731%	6,855%	4,773%
bier127.tsp	118282	15,324%	6,149%	11,234%	3,515%
ch130.tsp	6110	24,157%	4,157%	16,678%	3,912%
d1655.tsp	62128		41,835%		
d198.tsp	15780	18,916%	9,867%	2,205%	1,502%
d657.tsp	48912	52,327%	25,386%		5,655%
eil101.tsp	629	18,76%	3,975%	18,919%	3,657%
eil51.tsp	426	7,981%	1,643%	10,094%	0,469%
eil76.tsp	538	12,082%	1,673%	14,312%	0,929%
fl1400.tsp	20127		20,296%		
gil262.tsp	2378	42,473%	11,018%	18,713%	5,887%
kroA100.tsp	21282	19,002%	1,424%	19,444%	3,181%
kroA150.tsp	26524	30,772%	4,064%	19,175%	5,934%
pr1002.tsp	259045		29,658%		
u724.tsp	41910		33,102%		

Table 6.3: The deviation of the different applications from the optimal distance.

Looking at table 6.3 the deviations from the optimal solutions can be seen, giving a better view of how close we actually are compared to the other two programs. The SimpleACS is 4-18 percentage points better than our non-optimized program except for the d198 instance, where we actually have an improvement of app. 7 percentage points. With Chirico's original program it's a different story; here we are between 0,5 and 24 percentage points better, except for the kroA100 and eilXXX instances, where he is between app. 0,2 and 2 percentage points better. So even if we at times get worse results that Chirico's program, our improved instances compensate greatly. On the other hand we are lacking in ability to come up with as many solutions as Chirico, and as seen back in table 6.1 we still spend extra time on getting to our results. In figure 6.7 we have put the three programs' performances into a chart, where we are using our performance as Index 100, so that it is possible to see the others' performance compared to ours. In figure 6.8 the time consumption of the programs can be seen with our application as index 100.

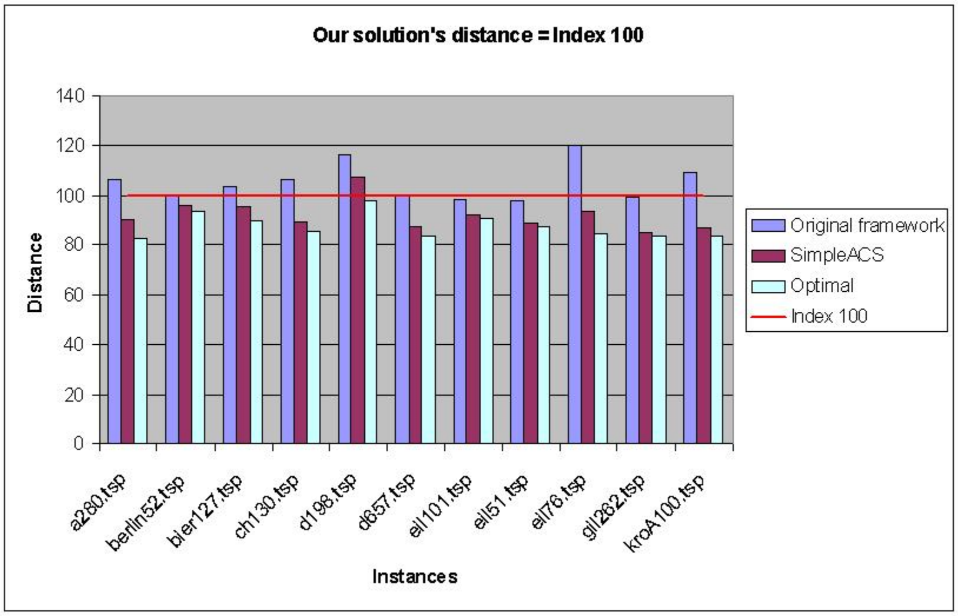


Figure 6.7: Our un-optimized preformance as index 100 compared to the optimal solution, Chirico's application and SimpleACS.

Looking at the deviations of our program with optimization activated, the differences in the deviations between our and the other programs are more striking, considering that we even get better results than most of those of the SimpleACS; only three instances (berlin52, kroA100 and kroA150) are solved better by SimpleACS than by ours (if one chooses to disregard the fact that SimpleACS is able to handle four instances that ours can not), with a difference that varies between 1,5 and 2 percentage points. Those we solve better are in an interval from app. 0,2 to 20 percentage points. Comparing with Chirico's original program we are only getting better results than his, all ranging from 3 to 47 percentage points. Unlike when running the program without the optimization, we also have a great time saving compared to his. On figure 6.9 the performances and optimal solution can be seen with ours as index 100. As it can be seen in 6.10 our time consumption has dropped greatly, being at par - if not better - with SimpleACS.

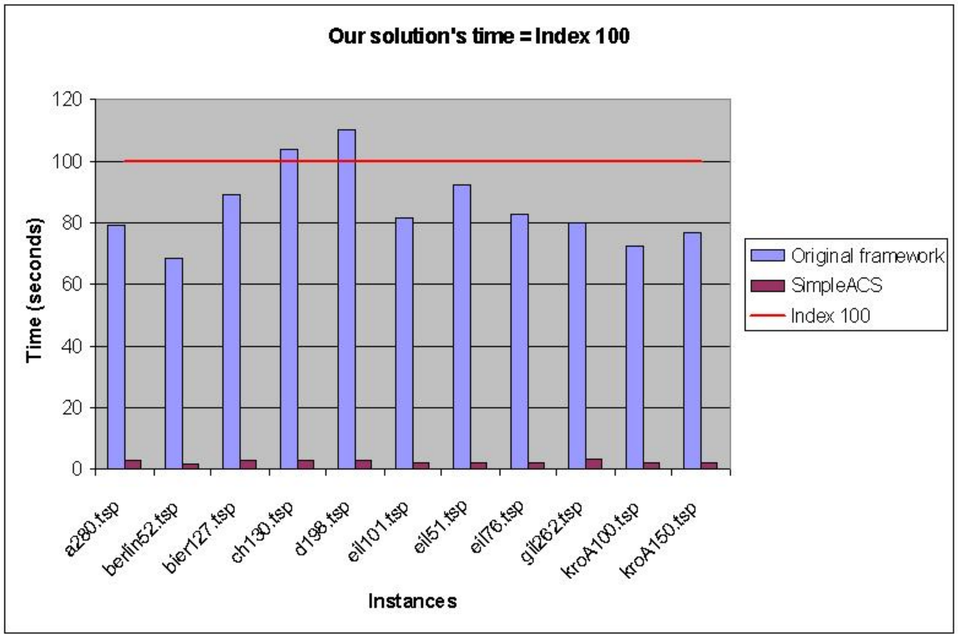


Figure 6.8: Our time consumption without optimization as index 100 compared to Chirico's application and SimpleACS.

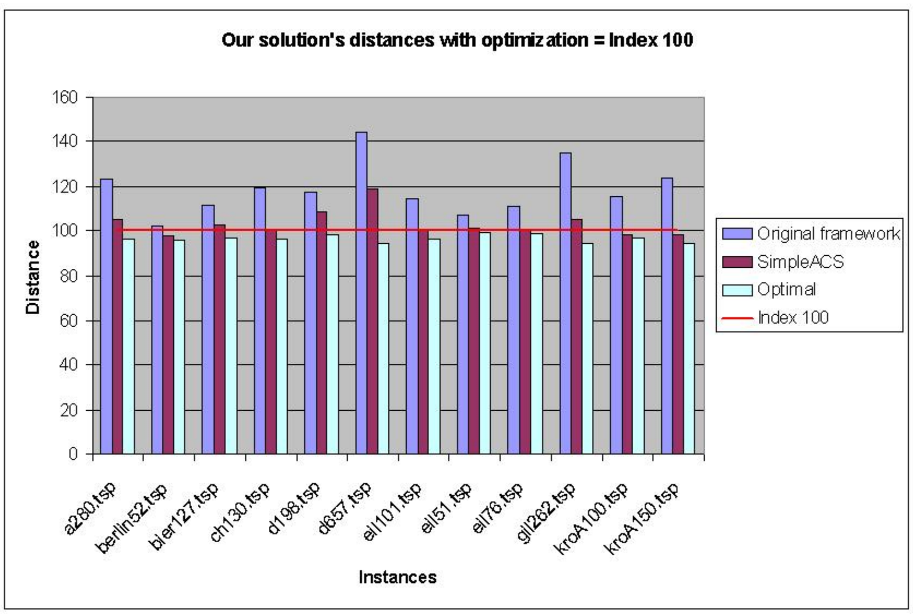


Figure 6.9: Our optimized preformance as index 100 compared to the optimal solution, Chirico's application and SimpleACS.

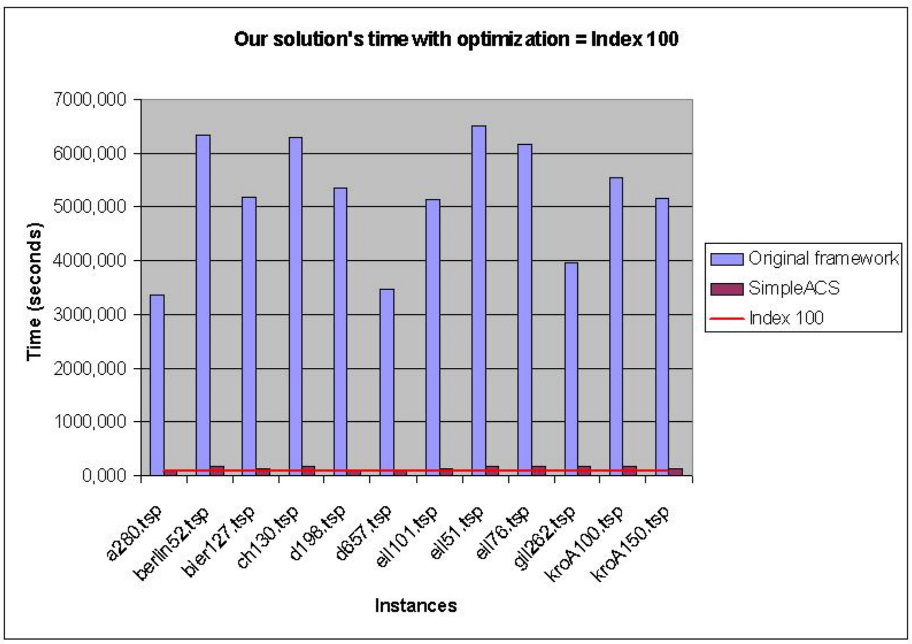


Figure 6.10: Our time consumption with optimization as index 100 compared to Chirico's application and SimpleACS.

7 Discussion

Taking our assumption from the introduction, one could question the reasons why we chose to apply the changes to the original framework. By making changes on the framework we are giving it settings so that several applications can be easily implemented without ruining its ability to solve a combinatorial problem using ants. We changed e.g. the graph because when using the graph we wanted to keep track of the nodes' coordinates. However, it could have been possible to have only the node IDs as integers as it would have lowered the required space and possibly sped up the algorithms. Meanwhile, we are convinced that it is better to use a node structure as it contains necessary information, as we accessed the node list to compute the nearest neighbor tour length used by the graph to compute τ_0.

We considered implementing a framework for the TSP in the ACO framework, but we realised that having a framework for the TSP which is using an ACO structure is very similar to an ACO framework. The most important methods needed by a TSP framework are the pheromone updating rules and the state transition rule. Both are already defined in the ACO framework, so building a framework for the TSP would be like building the ACO framework once again.

We are not convinced that the framework can handle all ant colony algorithms, as we found out that the computation for η (see chapter 3) is not the same for the TSP and QAP[10]. In fact, our heurisitic desirability is expressed as the inverse of a distance matrix, whereas in the QAP it is expressed as the heuristic matrix made from the product of the distance and flow matrices. A way to handle this is to override the graph object in the framework by a new method that will be required for the computation of such heuristic desirability.

From the above analysis, we are still questioning ourselves about using a framework instead of a specialized implementation in terms of speed. As the figures 6.8 and 6.10 show, if one's primary focus is on the speed of computation, then a framework is not the best choice. Instead one should focus on creating a specialized implementation with a single goal in mind. An example is an application of a framework on a routing system, the figures 6.8 and 6.10 will be able to give an idea of what could happen to the speed of the transmissions on the network.

However, one shouldn't neglect the importance a framework has as a data structure, which can be used to model other kinds of data.

[10] The quadratic assignment problem.

8 Conclusion

We took Chirico's original program and changed it so it supports node objects instead of integers, and we applied a few changes in the structure based on Dorigo's implementation of the TSP. Unfortunately we did not have time for testing different combinations of the program's parameters, trying to find a golden combination that would give us a a better result than other combinations, and instead we decided to go with values recommended by Dorigo. This resulted in slightly better results than Chirico's original program with about half of the tested cases, however at the cost of longer run times. Our results deviated 2-20% from the optimal solutions.

When applying the 2-opt and 2,5-opt algorithms, we made the program decrease its computation times with over 97% while improving the results by 0,6-13,9%. These results were all better than Chirico's original program, and compared to a specialized program not based on a framework, most of the results were better or similar. Our results when using optimization deviated from less than 0,5% to 6%, unfortunately they never reached the optimal solution. The time spent calculating the results were on a par with those of the specialized program, while the times for Chirico's program were now 35-65 times higher.

We added a GUI which gave the user an easy-to-use interface, where he or she could find and choose what instance he wanted to test, and change the parameters. We also added a graphical view of the tour, giving the user the opportunity to see how a tour looked when it had been created.

Consequently we feel that we have answered satisfactory on our problem formulation; our test results show that our program is slightly better than Chirico's when not using optimization, and when using optimization, the program competes well with other TSP implementations.

9 Perspectives

We feel that using Chirico's original application by expanding it with TSP and optimization algorithms, its status as a framework has been upheld. The changes made in Chirico's original version are mostly directed inwards at its internal structure, not affecting the public access points, albeit some of them have been undergoing changes to support the use of `Node` objects. We are quite satisfied with what we have done with the program, but nothing is perfect, so if our application should have had an overhaul done, several possibilities would be available:

- Expanding the GUI's functionality, so that it shows - or is able to show if the user wants it - more information on the progress of a tour construction, so that it is possible to see how far the computation is.
- Changing the structure of the optimization algorithms, so they look more like our pseudo code as shown on page 17 and 19. We believe that by doing this we will recieve a great achievement in computation time, as the lists we are using now are very heavy.
- Implementing support for more than just the *EUC_2D* cases. This could be done by adding functionality into the `InputFile` class that checked the file's type and reacted accordingly.
- Removing the graph from the framework, so instead of having a distance matrix, the distances between nodes are calculated when needed. The `Node` class already has the required functionality to compute distances on the fly, an ability that outdates the graph class as we know it. Instead the graph should be connected to the `Node` class, coordinating and serializing calculation of distances. This results in that the `Ant` and `AntColony` are the only two classes included in the ACO framework. This we believe will decrease the time consumption.

The framework already has the basics done for working on a network, as the graph is serialized and the ants are threads. But moving the framework from a single machine to a large network would still require a certain amount of tweaking and adjustment, and it would also require implementation of various network interfaces (sockets and/or RMI). Depending on the effectiveness of the network implementation and the bandwidth of the network itself, it might be possible to achieve better results than those we have because of the greater amount of available computing power. Doing this for only small instances will probably not be worth the trouble, as the network bandwidth will be the major bottleneck, so the goal of the network implementation would be able to solve very large instances. This could be done by splitting up the instances into smaller ones, thereby giving each involved computer a small(er) instance to solve. But this would then require implementation of algorithms that optimally can split and combine the instances without loss of distance quality.

As an addition to the existing optimization algorithms, it could be interesting to apply a 3-opt or Lin-Kernighan algorithm to see how they would perform on the framework.

Bibliography

[BDDW02] Mikkel Bundgaard, Troels C. Damgaard, Federico Decara, and Jacob W. Winther. Ant routing system, 2002. Can be found together with its code on http://www.itu.dk/people/mikkelbu/projects/ars.html.

[BDT99] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. *Swarm Intelligence - From Natural to Artificial Systems*, chapter 2. Oxford University Press, 1999.

[BT00] Eric Bonabeau and Guy Théraulaz. Swarm smarts. *Scientific American, Inc.*, pages 72–79, March 2000.

[Chi04] Ugo Chirico. A java framework for ant colony systems. Technical report, Siemens Informatica S.p.A, 2004.

[DG97] Marco Dorigo and Luca Maria Gambardella. Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1), 1997.

[DS04] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization*. The MIT Press, 2004.

[ERV06] Matthias Englert, Heiko Röglin, and Berthold Vöcking. Worst case and probabilistic analysis of the 2-opt algorithm for the TSP. Technical report, Department of Computer Science, RWTH Aachen, Germany, July 2006. We used a shortened version of the document, app. 10 pages compared to the full version's 50.

[FJMO95] M. L. Fredman, D. S. Johnson, L. A. McGeoch, and G. Ostheimer. Data structures for traveling salesmen. *J. Algorithms*, (18):432–479, 1995.

[GADP89] S. Goss, S. Aron, J. L. Deneubourg, and J. M. Pasteels. Self-organized shortcuts in the argentine ant. *Naturwissenschaften*, (76):579–581, 1989.

[Hel98] Keld Helsgaun. An effective implementation of the lin-kernighan traveling salesman heuristic. Technical report, Roskilde University, 1998. From Datalogiske Skrifter (Writings on Computer Science), No. 81, 1998.

[JM97] David S. Johnson and Lyle A. McGeoch. The traveling salesman problem: A case study in local optimization. In *Local Search in Combinatorial Optimization*, pages 215–310. John Wiley and Sons, London, 1997. A Preliminary version of the chapter in the book written 1995.

[JM03] Uffe Thomas Volmer Jankvist and Magnus Kaas Meinild. Myreintelligens, 2003.

[Nil03] Christian Nilsson. Heuristics for the traveling salesman problem. Technical report, Linköping University, 2003.

[OMI99] Hiroyuki Okano, Shinji Misono, and Kazuo Iwano. New TSP construction heuristics and their relationships to the 2-opt. *Journal of Heuristics*, (5):71–88, 1999.

[Rei95] Gerhard Reinelt. Tsplib 95. Technical report, Institut für Angewandte Mathematik, 1995.

[Sed90] Robert Sedgewick. *Algorithms in C++*. Addison-Wesley Publishing Company, 1990.

[ZMH06] Salah Zidi, Salah Maouche, and Slim Hammadi. Ant colony with dynamic local search for the time scheduling of transport networks. *International Journal of Computers, Communications & Control*, 1(4):110–125, 2006.

A Screenshots

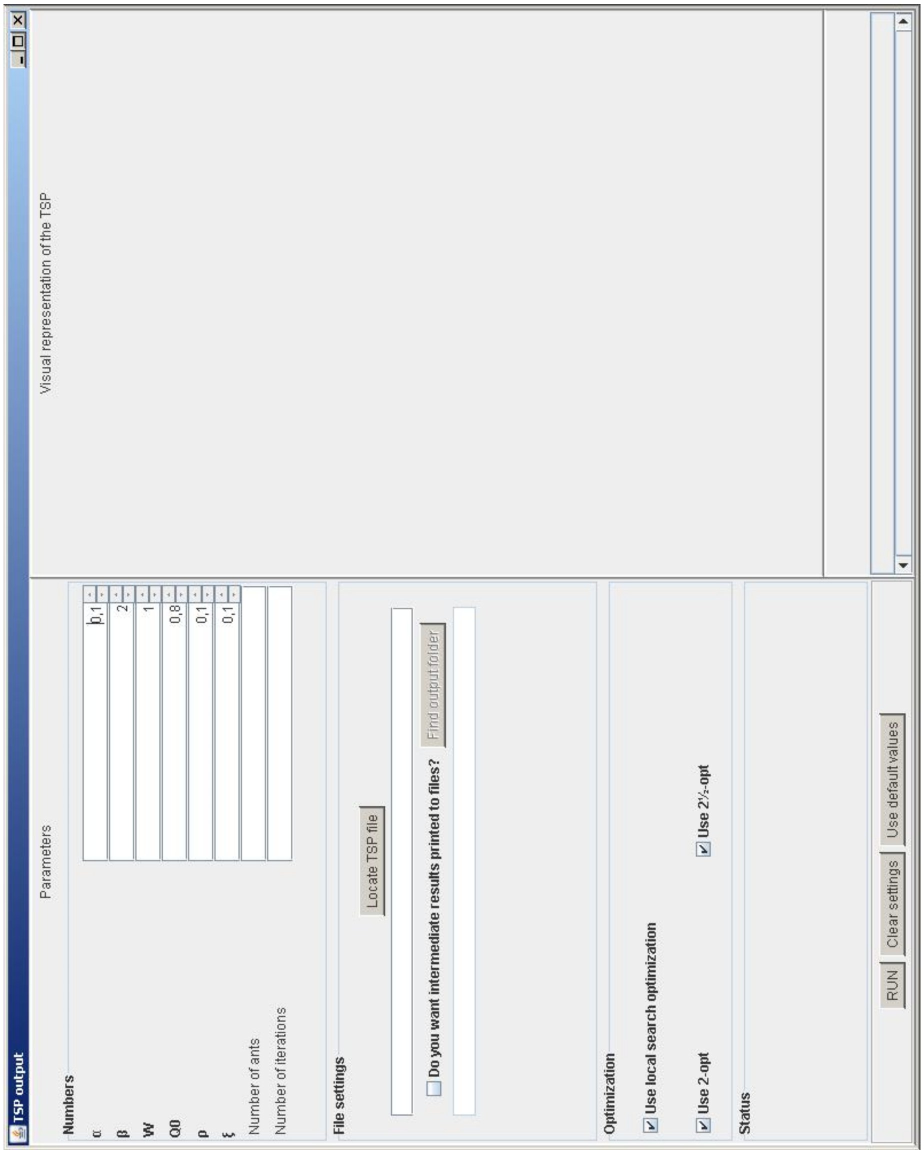


Figure A.1: The main window when the program is started.

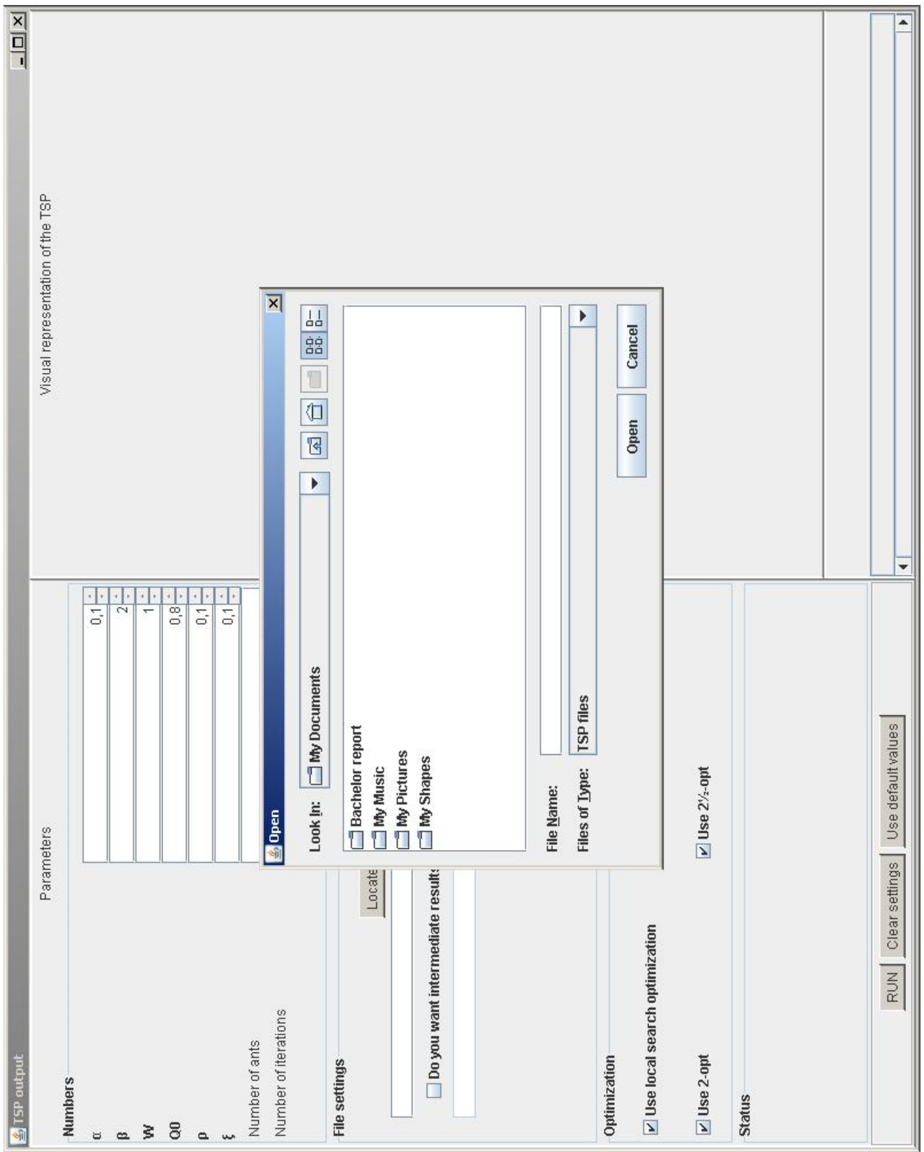


Figure A.2: The open file dialog box when locating an instance to run.

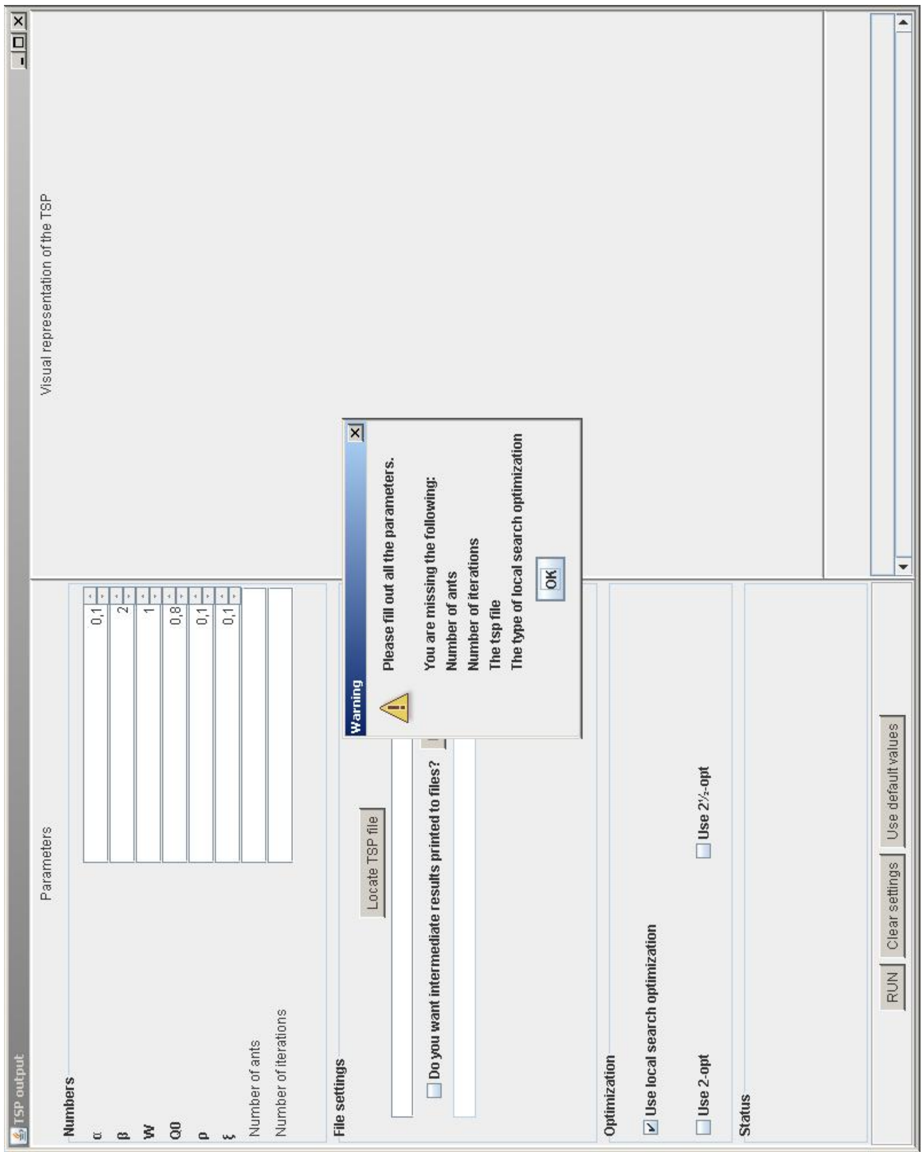


Figure A.3: The program found an error in the input.

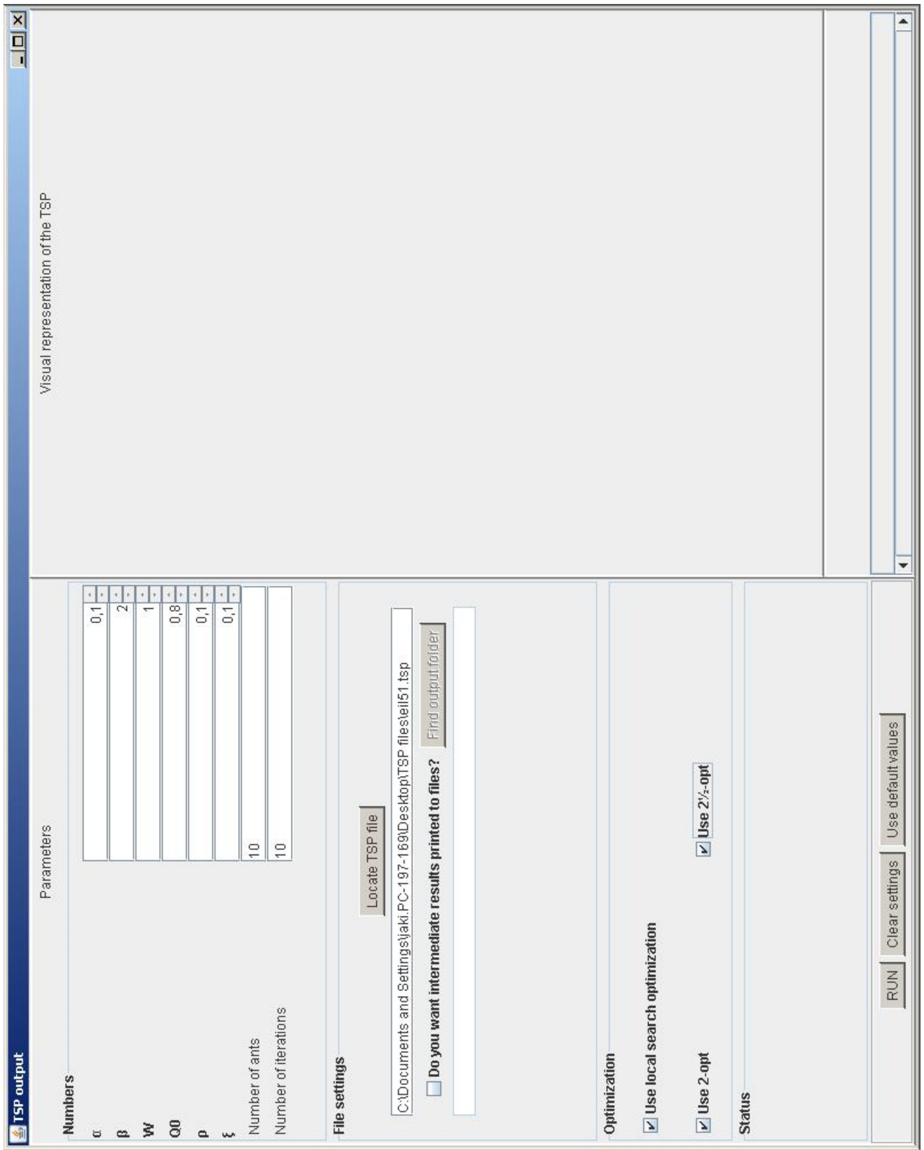


Figure A.4: The program when it's ready to run.

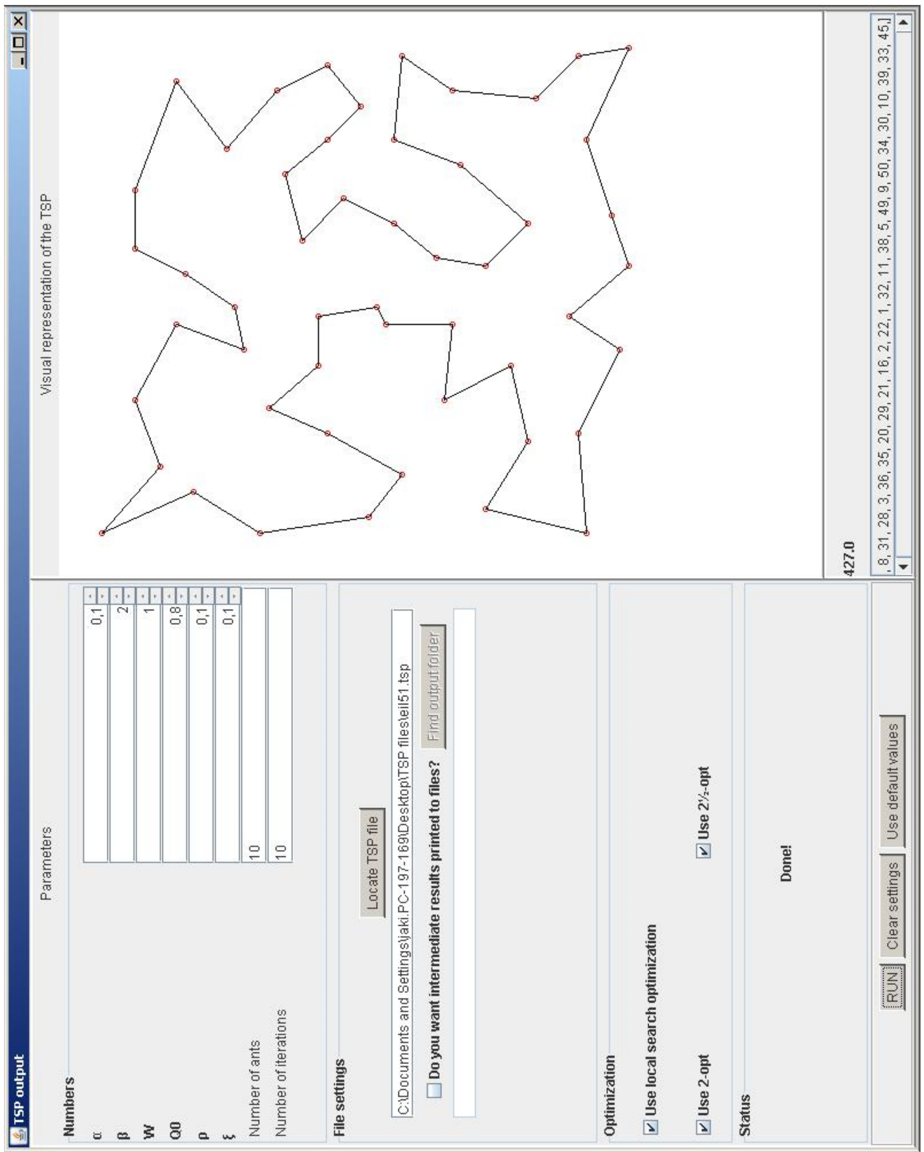


Figure A.5: The program showing the user the tour.

B Contents on the CD

We have enclosed a CD containing a few things we felt would be convenient for the reader to have access to.

The root consists of four folders; Report, JavaDoc, Code and TSPFiles.

- The Report folder contains the report in a PDF file format.
- The JavaDoc folder provides the Java documentation written in our application's code.
- The folder TSPFiles contains the tsp files of the type Eucledian 2D which we have used for benchmarking.
- The code folder contains a TSP.jar file (which is our application), and three other folders; TSP, SimpleACS and Chirico.
  - The TSP folder contains our program, including both java and class files.
  - The SimpleACS folder contains the SimpleACS program.
  - In the Chirico folder you can find Chirico's original program.

C Code for SimpleACS

This program has been handed to us by our supervisor, which has been taken from a former project he has been supervising.

C.1 Loader.java

```java
import java.util.*;
import java.io.*;

public class Loader {

  int numCities = 0;
  String workStr = "";

  public int[][] loadData(String path)
  {
    StringTokenizer strTok;
    if(path != null)
    {
      BufferedReader reader;
      try
      {
        reader = new BufferedReader(new FileReader(path));
        while(reader.ready())
        workStr += reader.readLine()+"\n";
      }
      catch(FileNotFoundException e)
      {
        System.err.println(e +
                   "\nFilen blev ikke fundet. " +
                   "Indtast nyt filnavn og pr v igen.");
      }
      catch(IOException e)
      {
        System.err.println(e +
                   "\nHardwarefejl under l sning. " +
                   "Indtast nyt filnavn og pr v igen.");
      }
    }
    numCities = getNumberOfCities(new StringTokenizer(workStr, " \n\t\r\f"));
    return buildDistMatrix(new StringTokenizer(workStr, " \n\t\r\f"));
  }

  private int getNumberOfCities(StringTokenizer strTok)
  {
    String tempStr = "";
    while(true)
    {
      tempStr = strTok.nextToken();
      if( tempStr.equals("DIMENSION"))
      {
        strTok.nextToken();
        tempStr = strTok.nextToken();
        return Integer.parseInt(tempStr);
      }
      else if(tempStr.equals("DIMENSION:"))
      {
        tempStr = strTok.nextToken();
        return Integer.parseInt(tempStr);
      }
    }
```

```
  }

  private int[][] buildDistMatrix(StringTokenizer strTok)
  {
    String tempStr = "";
    String edgeWeightType = "UNKNOWN";

    while(true)
    {
      tempStr = strTok.nextToken();
      if( tempStr.equals("EDGE_WEIGHT_TYPE"))
      {
        strTok.nextToken();
        tempStr = strTok.nextToken();
        edgeWeightType = tempStr;
        if(edgeWeightType.equals("EXPLICIT"))
        {
          return buildDistMatrixEXPLICIT(new StringTokenizer(
                             workStr, " \n\t\r\f"));
        }
        else
        {
          return buildDistMatrixEUC_2D(new StringTokenizer(
                             workStr, " \n\t\r\f"));
        }
      }
      else if(tempStr.equals("EDGE_WEIGHT_TYPE:"))
      {
        tempStr = strTok.nextToken();
        edgeWeightType = tempStr;
        if(edgeWeightType.equals("EXPLICIT"))
        {
          return buildDistMatrixEXPLICIT(new StringTokenizer(
                             workStr, " \n\t\r\f"));
        }
        else
        {
          return buildDistMatrixEUC_2D(new StringTokenizer(
                             workStr, " \n\t\r\f"));
        }
      }
    }
  }

  private int[][] buildDistMatrixEUC_2D(StringTokenizer strTok)
  {
    final int X = 0;
    final int Y = 1;
    int tempMatrix[][] = new int[numCities][numCities];
    String tempStr = "";
    double x,y = 0.0;
    int counter = 0;
    double coords[][] = new double[numCities][2];
    int dist = 0;
    while(true)
    {
      tempStr = strTok.nextToken();
      if( tempStr.equals("NODE_COORD_SECTION"))
      {

        while(!strTok.nextToken().equals("EOF"))
        {
          coords[counter][X] = Double.parseDouble(strTok.nextToken());
          coords[counter][Y] = Double.parseDouble(strTok.nextToken());
          counter++;
        }
        break;
      }
    }
    for(int j = 0; j < coords.length; j++)
    {
      for(int i = j; i < coords.length; i++)
      {
        dist = (int)Math.floor(.5 + Math.sqrt(
```

```
          Math.pow(coords[i][X] - coords[j][X],2.0) +
          Math.pow(coords[i][Y] - coords[j][Y],2.0)));

        tempMatrix[i][j] = dist;
        tempMatrix[j][i] = dist;
      }
    }
    return tempMatrix;
  }

  private int[][] buildDistMatrixEXPLICIT(StringTokenizer strTok)
  {
    int tempMatrix[][] = new int[numCities][numCities];
    int countI = 0;
    int countJ = 0;
    String tempStr = "";

    while(true)
    {
      tempStr = strTok.nextToken();
      if( tempStr.equals("EDGE_WEIGHT_SECTION"))
      {
        while(true)
        {
          tempStr = strTok.nextToken();
          if(tempStr.equals("EOF"))
          {
            return tempMatrix;
          }
          if(tempStr.equals("0"))
          {
            tempMatrix[countI][countJ] = Integer.parseInt(tempStr);
            tempMatrix[countJ][countI] = Integer.parseInt(tempStr);
            countI++;
            countJ=0;
          }
          else
          {
            tempMatrix[countI][countJ] = Integer.parseInt(tempStr);
            tempMatrix[countJ][countI] = Integer.parseInt(tempStr);
            countJ++;
          }
        }
      }
    }
  }
}
```

C.2 SimpleACS.java

```
import java.io.*;
import java.util.*;

public class SimpleACS {
    static final double BETA = 2,
                  GAMMA = 0.1,
                  qZERO = 0.9,
                  Q = 1.0;
  static final int M = 2,
              TMAX = 50000;
  static final Random random = new Random();

  int CITIES;
  double TAUZERO;
  int distances[][];
  double visibility[][];
  double pheromones[][];
  int bestTour[];
  int bestLength = Integer.MAX_VALUE;
  boolean tabu[];

```

```
  public static void main(String args[]) {
    Loader l = new Loader();
    SimpleACS main = new SimpleACS(l.loadData("eil51" + ".tsp"));
  }

  SimpleACS(int distances[][]) {
    this.distances = distances;
    CITIES = distances.length;
    bestTour = new int[CITIES];
    tabu = new boolean[CITIES];
    initialize();
    run();
    System.out.println(bestLength);
    for (int i = 0; i < CITIES + 1; i++)
      System.out.print(bestTour[i]+ " ");
  }

  public void initialize() {
    int NNTour[] = new int[CITIES + 1];
    pheromones = new double[CITIES][CITIES];
    visibility = new double[CITIES][CITIES];

    NNTour[0] = NNTour[CITIES] = 0;
    tabu[0] = true;
    for (int i = 1; i < CITIES; i++) {
      int nearest = 0;
      for (int j = 0; j < CITIES; j++)
        if (!tabu[j] &&
           (nearest == 0 ||
            distances[NNTour[i - 1]][j] < distances[i - 1][nearest]))
                nearest = j;
      NNTour[i] = nearest;
      tabu[nearest] = true;
    }
      bestTour = NNTour;
    bestLength = computeLength(NNTour);
    TAUZERO = 1.0 / (CITIES * bestLength);
    System.out.println("NN = " + bestLength);
    for (int i = 0; i < CITIES; i++)
      for (int j = 0; j < CITIES; j++)
        pheromones[i][j] = TAUZERO;
    for (int i = 0; i < CITIES; i++)
      for (int j = 0; j < CITIES; j++)
        visibility[i][j] =  Math.pow(distances[i][j], -BETA);
  }

  public void run() {
    for (int t = 0; t < TMAX; t++) {
        if (t % 100 == 0)
            System.out.println("Iteration " + t);
      for (int k = 0; k < M; k++)
        buildTour();
      for (int i = 0; i < CITIES; i++)
        pheromones[bestTour[i]][bestTour[i + 1]] =
        pheromones[bestTour[i + 1]][bestTour[i]] =
            (1 - GAMMA) * pheromones[bestTour[i]][bestTour[i + 1]] +
            GAMMA * (Q / bestLength);
    }
  }

  public void buildTour() {
    int tempTour[] = new int[CITIES + 1];
    int tempLength;
    double weights[] = new double[CITIES];
    double sigmaWeights;
    double q, tempWeight, target;
    int last, next;

    for (int i = 0; i < CITIES; i++)
      tabu[i] = false;
    last = tempTour[0] = tempTour[CITIES] = random.nextInt(CITIES);
    tabu[last] = true;
    for (int i = 1; i < CITIES; i++) {
      for (int j = 0; j < CITIES; j++)
```

```
            weights[j] = tabu[j] ? 0 :
              pheromones[last][j] * visibility[last][j];
      q = random.nextDouble();
      next = 0;
      if (q <= qZERO) {
        tempWeight = 0;
        for (int j = 0; j < CITIES; j++) {
          if (weights[j] > tempWeight) {
            tempWeight = weights[j];
            next = j;
          }
          }
      } else {
          sigmaWeights = 0;
        for (int j = 0; j < CITIES; j++)
          sigmaWeights += weights[j];
        target = random.nextDouble() * sigmaWeights;
        tempWeight = 0;
        for (int j = 0; j < CITIES; j++) {
          tempWeight += weights[j];
          if (tempWeight >= target) {
                  next = j;
                  break;
              }
          }
      }
      if (tabu[next]) { System.out.println("TABU\n" + next); System.exit(0); }
      pheromones[last][next] = pheromones[next][last] =
          (1 - GAMMA) * pheromones[last][next] + GAMMA * TAUZERO;
          tempTour[i] = last = next;
      tabu[last] = true;
    }
    tempLength = computeLength(tempTour);
    if (tempLength < bestLength) {
      bestTour = tempTour;
      bestLength = tempLength;
        System.out.println("Best = " + bestLength);
    }
  }

  int computeLength(int tour[]) {
    int length = 0;
    for (int i = 0; i < CITIES; i++)
      length += distances[tour[i]][tour[i+1]];
    return length;
  }
}
```

D Chirico's original program

D.1 Ant4TSP.java

```
/**
 * Ant4TSP.java
 *
 * @author Created by Omnicore CodeGuide
 */

package com.ugos.acs.tsp;
import com.ugos.acs.*;

import java.util.*;


public class Ant4TSP extends Ant
{
    private static final double B    = 2;
    private static final double Q0   = 0.8;
    private static final double R    = 0.1;

    private static final Random s_randGen = new Random(System.currentTimeMillis());

    protected Hashtable m_nodesToVisitTbl;

    public Ant4TSP(int startNode, Observer observer)
    {
        super(startNode, observer);
    }

    public void init()
    {
        super.init();

        final AntGraph graph = s_antColony.getGraph();

        // inizializza l'array di citt da visitare
        m_nodesToVisitTbl = new Hashtable(graph.nodes());
        for(int i = 0; i < graph.nodes(); i++)
            m_nodesToVisitTbl.put(new Integer(i), new Integer(i));

        // Rimuove la citt corrente
        m_nodesToVisitTbl.remove(new Integer(m_nStartNode));

//        nExplore = 0;
    }

    public int stateTransitionRule(int nCurNode)
    {
        final AntGraph graph = s_antColony.getGraph();

        // generate a random number
        double q    = s_randGen.nextDouble();
        int nMaxNode = -1;

        if(q <= Q0)  // Exploitation
        {
//            System.out.print("Exploitation: ");
            double dMaxVal = -1;
            double dVal;
            int nNode;

            // search the max of the value as defined in Eq. a)
            Enumeration enum1 = m_nodesToVisitTbl.elements();
```

```
            while(enum1.hasMoreElements())
            {
                // select a node
                nNode = ((Integer)enum1.nextElement()).intValue();

                // check on tau
                if(graph.tau(nCurNode, nNode) == 0)
                    throw new RuntimeException("tau = 0");

                // get the value
                dVal = graph.tau(nCurNode, nNode) * Math.pow(graph.etha(
                          nCurNode, nNode), B);

                // check if it is the max
                if(dVal > dMaxVal)
                {
                    dMaxVal  = dVal;
                    nMaxNode = nNode;
                }
            }
        }
        else  // Exploration
        {
//              System.out.println("Exploration");
            double dSum = 0;
            int nNode = -1;

            // get the sum at denominator
            Enumeration enum1 = m_nodesToVisitTbl.elements();
            while(enum1.hasMoreElements())
            {
                nNode = ((Integer)enum1.nextElement()).intValue();
                if(graph.tau(nCurNode, nNode) == 0)
                    throw new RuntimeException("tau = 0");

                // Update the sum
                dSum += graph.tau(nCurNode, nNode) * Math.pow(graph.etha(
                            nCurNode, nNode), B);
            }

            if(dSum == 0)
                throw new RuntimeException("SUM = 0");

            // get the everage value
            double dAverage = dSum / (double)m_nodesToVisitTbl.size();

            // search the node in agreement with eq. b)
            enum1 = m_nodesToVisitTbl.elements();
            while(enum1.hasMoreElements() && nMaxNode < 0)
            {
                nNode = ((Integer)enum1.nextElement()).intValue();

                // get the value of p as difined in eq. b)
                double p =
                    (graph.tau(nCurNode, nNode) * Math.pow(graph.etha(
                        nCurNode, nNode), B)) / dSum;

                // if the value of p is greater the the average value
        // the node is good
                if((graph.tau(nCurNode, nNode) * Math.pow(graph.etha(
                      nCurNode, nNode), B)) > dAverage)
                {
                    //System.out.println("Found");
                    nMaxNode = nNode;
                }
            }

            if(nMaxNode == -1)
                nMaxNode = nNode;
       }

        if(nMaxNode < 0)
            throw new RuntimeException("maxNode = -1");

```

```
            // delete the selected node from the list of node to visit
            m_nodesToVisitTbl.remove(new Integer(nMaxNode));

            return nMaxNode;
        }

        public void localUpdatingRule(int nCurNode, int nNextNode)
        {
            final AntGraph graph = s_antColony.getGraph();

            // get the value of the Eq. c)
            double val =
                ((double)1 - R) * graph.tau(nCurNode, nNextNode) +
                (R * (graph.tau0()));

            // update tau
            graph.updateTau(nCurNode, nNextNode, val);
        }

        public boolean better(double dPathValue1, double dPathValue2)
        {
            return dPathValue1 < dPathValue2;
        }

        public boolean end()
        {
            return m_nodesToVisitTbl.isEmpty();
        }
}
```

D.2 Ant.java

```
/**
 * Ant.java
 *
 * @author Created by Omnicore CodeGuide
 */

package com.ugos.acs;

import java.util.*;
import java.io.*;

public abstract class Ant extends Observable implements Runnable
{
    private int m_nAntID;

    protected int[][]  m_path;
    protected int      m_nCurNode;
    protected int      m_nStartNode;
    protected double   m_dPathValue;
    protected Observer m_observer;
    protected Vector   m_pathVect;

    private static int s_nAntIDCounter = 0;
    private static PrintStream s_outs;

    protected static AntColony s_antColony;

    public static double    s_dBestPathValue = Double.MAX_VALUE;
    public static Vector    s_bestPathVect  = null;
    public static int[][]   s_bestPath      = null;
    public static int       s_nLastBestPathIteration = 0;

    public static void setAntColony(AntColony antColony)
    {
        s_antColony = antColony;
    }

    public static void reset()
    {
```

```
        s_dBestPathValue = Double.MAX_VALUE;
        s_bestPathVect = null;
        s_bestPath = null;
        s_nLastBestPathIteration = 0;
        s_outs = null;
    }

    public Ant(int nStartNode, Observer observer)
    {
        s_nAntIDCounter++;
        m_nAntID     = s_nAntIDCounter;
        m_nStartNode = nStartNode;
        m_observer   = observer;
    }

    public void init()
    {
        if(s_outs == null)
        {
            try
            {
                s_outs = new PrintStream(new FileOutputStream("c:\\temp\\" +
            s_antColony.getID()+ "_" + s_antColony.getGraph().nodes() +
            "x" + s_antColony.getAnts() + "x" + s_antColony.getIterations() +
            "_ants.txt"));
            }
            catch(Exception ex)
            {
                ex.printStackTrace();
            }
        }

        final AntGraph graph = s_antColony.getGraph();
        m_nCurNode   = m_nStartNode;

        m_path       = new int[graph.nodes()][graph.nodes()];
        m_pathVect   = new Vector(graph.nodes());

        m_pathVect.addElement(new Integer(m_nStartNode));
        m_dPathValue = 0;
    }

    public void start()
    {
        init();
        Thread thread = new Thread(this);
        thread.setName("Ant " + m_nAntID);
        thread.start();
    }

    public void run()
    {
        final AntGraph graph = s_antColony.getGraph();

        // repeat while End of Activity Rule returns false
        while(!end())
        {
            int nNewNode;

            // synchronize the access to the graph
            synchronized(graph)
            {
                // apply the State Transition Rule
                nNewNode = stateTransitionRule(m_nCurNode);

                // update the length of the path
                m_dPathValue += (int)(graph.delta(m_nCurNode, nNewNode) + 0.5);
            }

            // add the current node the list of visited nodes
            m_pathVect.addElement(new Integer(nNewNode));
            m_path[m_nCurNode][nNewNode] = 1;

            synchronized(graph)
```

```
            {
                // apply the Local Updating Rule
                localUpdatingRule(m_nCurNode, nNewNode);
            }

            // update the current node
            m_nCurNode = nNewNode;
        }

        synchronized(graph)
        {
            // update the best tour value
            if(better(m_dPathValue, s_dBestPathValue))
            {
                s_dBestPathValue        = m_dPathValue;
                s_bestPath              = m_path;
                s_bestPathVect          = m_pathVect;
                s_nLastBestPathIteration = s_antColony.getIterationCounter();

                s_outs.println("Ant + " + m_nAntID + "," + s_dBestPathValue +
                "," + s_nLastBestPathIteration + "," +
                s_bestPathVect.size() + "," + s_bestPathVect);
            }
        }

        // update the observer
        m_observer.update(this, null);

        if(s_antColony.done())
            s_outs.close();
    }

    protected abstract boolean better(double dPathValue, double dBestPathValue);

    public abstract int stateTransitionRule(int r);

    public abstract void localUpdatingRule(int r, int s);

    public abstract boolean end();

    public static int[] getBestPath()
    {
        int nBestPathArray[] = new int[s_bestPathVect.size()];
        for(int i = 0; i < s_bestPathVect.size(); i++)
        {
            nBestPathArray[i] = ((Integer)s_bestPathVect.elementAt(i)).intValue();
        }

        return nBestPathArray;
    }

    public String toString()
    {
        return "Ant " + m_nAntID + ":" + m_nCurNode;
    }
}
```

D.3 AntColony4TSP.java

```
/**
 * Pure Java console application.
 * This application demonstrates console I/O.
 *
 * This file was automatically generated by
 * Omnicore CodeGuide.
 */
package com.ugos.acs.tsp;
import com.ugos.acs.*;

import java.util.*;
import java.io.*;
```

```java

public class AntColony4TSP extends AntColony
{
    protected static final double A = 0.1;

    public AntColony4TSP(AntGraph graph, int ants, int iterations)
    {
        super(graph, ants, iterations);
    }

    protected Ant[] createAnts(AntGraph graph, int nAnts)
    {
        Random ran = new Random(System.currentTimeMillis());
        Ant4TSP.reset();
        Ant4TSP.setAntColony(this);
        Ant4TSP ant[] = new Ant4TSP[nAnts];
        for(int i = 0; i < nAnts; i++)
        {
            ant[i] = new Ant4TSP((int)(graph.nodes() * ran.nextDouble()), this);
        }

        return ant;
    }

    protected void globalUpdatingRule()
    {
        double dEvaporation = 0;
        double dDeposition  = 0;

        for(int r = 0; r < m_graph.nodes(); r++)
        {
            for(int s = 0; s < m_graph.nodes(); s++)
            {
                if(r != s)
                {
                    // get the value for deltatau
                    double deltaTau = //Ant4TSP.s_dBestPathValue *
                    // (double)Ant4TSP.s_bestPath[r][s];
                        ((double)1 / Ant4TSP.s_dBestPathValue) *
                      (double)Ant4TSP.s_bestPath[r][s];

                    // get the value for phermone evaporation as defined in eq. d)
                    dEvaporation = ((double)1 - A) * m_graph.tau(r,s);
                    // get the value for phermone deposition as defined in eq. d)
                    dDeposition  = A * deltaTau;

                    // update tau
                    m_graph.updateTau(r, s, dEvaporation + dDeposition);
                }
            }
        }
    }
}
```

D.4 AntColony.java

```java
/**
 * Pure Java console application.
 * This application demonstrates console I/O.
 *
 * This file was automatically generated by
 * Omnicore CodeGuide.
 */

package com.ugos.acs;

import java.util.*;
import java.io.*;


public abstract class AntColony implements Observer
```

```
{
    protected PrintStream m_outs;

    protected AntGraph m_graph;
    protected Ant[]    m_ants;
    protected int      m_nAnts;
    protected int      m_nAntCounter;
    protected int      m_nIterCounter;
    protected int      m_nIterations;

    private int      m_nID;

    private static int s_nIDCounter = 0;

    public AntColony(AntGraph graph, int nAnts, int nIterations)
    {
        m_graph = graph;
        m_nAnts = nAnts;
        m_nIterations = nIterations;
        s_nIDCounter++;
        m_nID = s_nIDCounter;
    }

    public synchronized void start()
    {
        // creates all ants
        m_ants  = createAnts(m_graph, m_nAnts);

        m_nIterCounter = 0;
        try
        {
            m_outs = new PrintStream(new FileOutputStream("c:\\temp\\" + m_nID +
                "_" + m_graph.nodes() + "x" + m_ants.length + "x" +
                m_nIterations + "_colony.txt"));
        }
        catch(Exception ex)
        {
            ex.printStackTrace();
        }

        // loop for all iterations
        while(m_nIterCounter < m_nIterations)
        {
            // run an iteration
            iteration();
            try
            {
                wait();
            }
            catch(InterruptedException ex)
            {
                ex.printStackTrace();
            }

            // synchronize the access to the graph
            synchronized(m_graph)
            {
                // apply global updating rule
                globalUpdatingRule();
            }
        }

        if(m_nIterCounter == m_nIterations)
        {
            m_outs.close();
        }
    }

    private void iteration()
    {
        m_nAntCounter = 0;
        m_nIterCounter++;
        m_outs.print(m_nIterCounter);
        for(int i = 0; i < m_ants.length; i++)
```

```
        {
            m_ants[i].start();
        }
    }

    public AntGraph getGraph()
    {
        return m_graph;
    }

    public int getAnts()
    {
        return m_ants.length;
    }

    public int getIterations()
    {
        return m_nIterations;
    }

    public int getIterationCounter()
    {
        return m_nIterCounter;
    }

    public int getID()
    {
        return m_nID;
    }

    public synchronized void update(Observable ant, Object obj)
    {
        //m_outs.print(";" + ((Ant)ant).m_dPathValue);
        m_nAntCounter++;

        if(m_nAntCounter == m_ants.length)
        {
            m_outs.println(";" + Ant.s_dBestPathValue + ";" + m_graph.averageTau());

            //  System.out.println("---------------------------");
            //  System.out.println(m_iterCounter + " - Best Path: " +
      //                      Ant.s_dBestPathValue);
            //  System.out.println("---------------------------");



            notify();

        }
    }

    public double getBestPathValue()
    {
        return Ant.s_dBestPathValue;
    }

    public int[] getBestPath()
    {
        return Ant.getBestPath();
    }

    public Vector getBestPathVector()
    {
        return Ant.s_bestPathVect;
    }

    public int getLastBestPathIteration()
    {
        return Ant.s_nLastBestPathIteration;
    }

    public boolean done()
    {
        return m_nIterCounter == m_nIterations;
```

```
    }

    protected abstract Ant[] createAnts(AntGraph graph, int ants);

    protected abstract void globalUpdatingRule();
}
```

D.5 AntGraph.java

```
/**
 * AntGraph.java
 *
 * @author Created by Omnicore CodeGuide
 */
package com.ugos.acs;

import java.io.*;

public class AntGraph implements Serializable
{
    private double[][] m_delta;
    private double[][] m_tau;
    private int        m_nNodes;
    private double     m_dTau0;

    public AntGraph(int nNodes, double[][] delta, double[][] tau)
    {
        if(delta.length != nNodes)
            throw new IllegalArgumentException("The number of nodes doesn't match
                      with the dimension of delta matrix");

        m_nNodes = nNodes;
        m_delta = delta;
        m_tau   = tau;
    }

    public AntGraph(int nodes, double[][] delta)
    {
        this(nodes, delta, new double[nodes][nodes]);

        resetTau();
    }

    public synchronized double delta(int r, int s)
    {
        return m_delta[r][s];
    }

    public synchronized double tau(int r, int s)
    {
        return m_tau[r][s];
    }

    public synchronized double etha(int r, int s)
    {
        return ((double)1) / delta(r, s);
    }

    public synchronized int nodes()
    {
        return m_nNodes;
    }

    public synchronized double tau0()
    {
        return m_dTau0;
    }

    public synchronized void updateTau(int r, int s, double value)
    {
        m_tau[r][s] = value;
```

```
    }

    public void resetTau()
    {
        double dAverage = averageDelta();

        m_dTau0 = (double)1 / ((double)m_nNodes * (0.5 * dAverage));

        System.out.println("Average: " + dAverage);
        System.out.println("Tau0: " + m_dTau0);

        for(int r = 0; r < nodes(); r++)
        {
            for(int s = 0; s < nodes(); s++)
            {
                m_tau[r][s] = m_dTau0;
            }
        }
    }

    public double averageDelta()
    {
        return average(m_delta);
    }

    public double averageTau()
    {
        return average(m_tau);
    }

    public String toString()
    {
        String str = "";
        String str1 = "";


        for(int r = 0; r < nodes(); r++)
        {
            for(int s = 0; s < nodes(); s++)
            {
                str += delta(r,s) + "\t";
                str1 += tau(r,s) + "\t";
            }

            str+= "\n";
        }

        return str + "\n\n\n" + str1;
    }

    private double average(double matrix[][])
    {
        double dSum = 0;
        for(int r = 0; r < m_nNodes; r++)
        {
            for(int s = 0; s < m_nNodes; s++)
            {
                dSum += matrix[r][s];
            }
        }

        double dAverage = dSum / (double)(m_nNodes * m_nNodes);

        return dAverage;
    }
}
```

D.6 TSPTest.java

```
/**
 * AntApplication.java
```

```java
 *
 * @author Created by Omnicore CodeGuide
 */

import java.util.*;
import java.io.*;
import com.ugos.acs.tsp.*;
import com.ugos.acs.*;

public class TSPTest
{
    private static Random s_ran = new Random(System.currentTimeMillis());

    public static void main(String[] args)
    {
        // Print application prompt to console.
        System.out.println("AntColonySystem for TSP");

        if(args.length < 8)
        {
            System.out.println("Wrong number of parameters");
            return;
        }

        int nAnts = 0;
        int nNodes = 0;
        int nIterations = 0;
        int nRepetitions = 0;

        for (int i = 0; i < args.length; i+=2)
        {
            if(args[i].equals("-a"))
            {
                nAnts = Integer.parseInt(args[i + 1]);
                System.out.println("Ants: " + nAnts);
            }
            else if(args[i].equals("-n"))
            {
                nNodes = Integer.parseInt(args[i + 1]);
                System.out.println("Nodes: " + nNodes);
            }
            else if(args[i].equals("-i"))
            {
                nIterations = Integer.parseInt(args[i + 1]);
                System.out.println("Iterations: " + nIterations );
            }
            else if(args[i].equals("-r"))
            {
                nRepetitions = Integer.parseInt(args[i + 1]);
                System.out.println("Repetitions: " + nRepetitions);
            }
        }

        if(nAnts == 0 || nNodes == 0 || nIterations == 0 || nRepetitions == 0)
        {
            System.out.println("One of the parameters is wrong");
            return;
        }


        double d[][] = new double[nNodes][nNodes];
//        double t[][] = new double[nNodes][nNodes];

        for(int i = 0; i < nNodes; i++)
            for(int j = i + 1; j < nNodes; j++)
            {
                d[i][j] = s_ran.nextDouble();
                d[j][i] = d[i][j];
//                t[i][j] = 1; //(double)1 / (double)(nNodes * 10);
//                t[j][i] = t[i][j];
            }

         AntGraph graph = new AntGraph(nNodes, d);

```

```
        try
        {
          ObjectOutputStream outs = new ObjectOutputStream(
                new FileOutputStream("c:\\temp\\" +
                nNodes + "_antgraph.bin"));
          outs.writeObject(graph);
          outs.close();

//            ObjectInputStream ins = new ObjectInputStream(
//                new FileInputStream("c:\\temp\\" +
//                nNodes + "_antgraph.bin"));
//            graph = (AntGraph)ins.readObject();
//            ins.close();

            FileOutputStream outs1 = new FileOutputStream("c:\\temp\\" +
                      nNodes + "_antgraph.txt");

            for(int i = 0; i < nNodes; i++)
            {
                for(int j = 0; j < nNodes; j++)
                {
                    outs1.write((graph.delta(i,j) + ",").getBytes());
                }
                outs1.write('\n');
            }

            outs1.close();

            PrintStream outs2 = new PrintStream(new FileOutputStream("c:\\temp\\" +
            nNodes + "x" + nAnts + "x" + nIterations + "_results.txt"));

            for(int i = 0; i < nRepetitions; i++)
            {
                graph.resetTau();
                AntColony4TSP antColony = new AntColony4TSP(graph, nAnts, nIterations);
                antColony.start();
                outs2.println(i + "," + antColony.getBestPathValue() + "," +
                      antColony.getLastBestPathIteration());
            }
            outs2.close();
        }
        catch(Exception ex)
        {}
    }
}
```

E Code

E.1 AntColonyForTSP.java

```
package tsp;

import framework.acos.*;
import java.util.*;
import Node.Node;

/**
 * The ant colony for a TSP solution.
 * Extends the ant colony from the ACS framework, and defines the
 * global updating rule and the way the ants are created.
 * @author Jakob Kierkegaard & Jean-Luc Ngassa
 */
public class AntColonyForTSP extends AntColonyFramework {
  /**
   * The rho value, received from the input in the GUI by the user.
   */
  protected static final double RHO = Main.GUI.getRho();
  /**
   * The W value, received from the input in the GUI by the user.
   */
  protected static final int W = Main.GUI.getW();
  /**
   * An array containing the best path yet.
   */
  int[] bestPath = new int[m_graph.nodes()];

  /**
   * The constructor calls its superclass' constructor using the
   * given parameters.
   * @param graph The graph the ants are to use.
   * @param ants The number of ants.
   * @param iterations The number of iterations
   */
  public AntColonyForTSP(AntGraphFramework graph, int ants, int iterations) {
    super(graph, ants, iterations);
  }

  /**
   * This method creates all the ants for the program.
   * It resets the best values, defines the ant colony, and destributes
   * the ants evenly out on the nodes (no randomness; the starting node
   * is antID%numberOfNodes). The ants are returned in an array.
   * @param graph The graph the ants are using.
   * @param nAnts The number of ants.
   * @return Returns an array of objects of the type AntForTSP; the
   * ants that are to find the tours on the graph.
   */
  protected AntFramework[] createAnts(AntGraphFramework graph, int nAnts) {
    AntForTSP.reset();
    AntForTSP.setAntColony(this);
    AntForTSP ant[] = new AntForTSP[nAnts];
    for (int i = 0; i < nAnts; i++)
      ant[i] = new AntForTSP((i % graph.nodes()), this);
    return ant;
  }

  /**
   * The global updating rule.
   * The algorithms for calculating the evaporation and deposition of
   * the pheromone on the edges that are part of the best tour so far,
   * reinforcing its edges.
```

```
   */
  protected void globalUpdatingRule() {
    double dEvaporation;
    double dDeposition;
    bestPath = AntForTSP.getBestPath();

    for (int r = 0; r < m_graph.nodes(); r++) {
      for (int s = r + 1; s < m_graph.nodes(); s++) {
        double deltaTau = (W / AntForTSP.s_dBestPathLength);

        dEvaporation = ((double) 1 - RHO) * m_graph.tau(bestPath[r], bestPath[s]);

        dDeposition = RHO * deltaTau;

        // update tau
        m_graph.updateTau(bestPath[r], bestPath[s], dEvaporation + dDeposition);
        m_graph.updateTau(bestPath[s], bestPath[r], dEvaporation + dDeposition);
      }
    }
  }

  /**
   * Retrieves the best tour represented by a LinkedList of node objects.
   * @return A linked list of node objects.
   */
  public LinkedList<Node> getBestTourNodes() {
    return nodesTour(AntFramework.s_bestTour);
  }

  /**
   * Returns the best tour represented by an ArrayList of integers
   * as the nodes' IDs.
   * @return An ArrayList of integers.
   */
  public ArrayList<Integer> getBestTour() {
    return tour(AntFramework.s_bestTour);
  }

  /**
   * Converts an ArrayList of integers (node IDs) to a LinkedList of
   * node objects. It does this by going through the ArrayList, and if
   * it finds a node with a similar ID, it creates a new node and
   * inserts it into the LinkedList being returned.
   * @param tour An ArrayList of integers.
   * @return A LinkedList of node objects.
   */
  public LinkedList<Node> nodesTour(ArrayList<Integer> tour) {
    LinkedList<Node> myTour = new LinkedList<Node>();

    for (Integer i : tour) {
      for (Node n : m_graph.getNodeList()) {
        if (n.nodeID + 1 == i)
          myTour.add(new Node(n.nodeID + 1, n.x, n.y));
      }
    }
    return myTour;
  }

  /**
   * In the program the node IDs are going from 0, but users need to
   * see them going from 1. This method adds one to all the node IDs.
   * @param tour The ArrayList with the Integers that are to be increased.
   * @return An ArrayList with the node IDs going from 1.
   */
  private static ArrayList<Integer> tour(ArrayList<Integer> tour) {
    for (int i = 0; i < tour.size(); i ++) {
      tour.set(i, tour.get(i) + 1);
    }
    return tour;
  }
}
```

E.2 AntColonyFramework.java

```java
package framework.acos;

import java.util.*;
import java.io.*;

/**
 * The AntColony keeps track of the ants on the graph, starting the
 * ants for each iteration, and applies the global updating rule.
 * As the ants are threads, the ant colony implements the Observer interface.
 *
 * @author Jakob Kierkegaard & Jean-Luc Ngassa
 */
public abstract class AntColonyFramework implements Observer {

  /**
   * The printstream if the user wants the intermediate results written
   * to files.
   */
  private PrintStream m_outs;
  /**
   * The graph in use.
   */
  protected final AntGraphFramework m_graph;
  /**
   * An array of all the ants.
   */
  protected AntFramework[] m_ants;
  /**
   * The number of ants.
   */
  protected final int m_nAnts;
  /**
   * The ant counter.
   */
  protected int m_nAntCounter;
  /**
   * The iteration counter.
   */
  protected int m_nIterCounter;
  /**
   * The number of iterations.
   */
  protected final int m_nIterations;
  /**
   * The ID of the colony.
   */
  private final int m_nID;
  /**
   * The colony counter - only relevant if more than one colony is
   * used, which we do not.
   */
  private static int s_nIDCounter = 0;

  /**
   * Initializes the local values required for running the ant colony.
   *
   * @param graph     The graph used.
   * @param nAnts     The number of ants used.
   * @param nIterations The number of iterations used.
   */
  public AntColonyFramework(AntGraphFramework graph, int nAnts, int nIterations) {
    m_graph = graph;
    m_nAnts = nAnts;
    m_nIterations = nIterations;
    s_nIDCounter++;
    m_nID = s_nIDCounter;
  }

  /**
   * Creates the ants, and after each iteration the global updating rule
   * is applied to the edges on the graph.
```

```
   */
  public synchronized void start() {
    // creates all ants
    m_ants = createAnts(m_graph, m_nAnts);

    m_nIterCounter = 0;

    if (Main.GUI.makeOutput()) {
      try {
        m_outs = new PrintStream(new FileOutputStream(Main.GUI.getOutputPath() +
              m_nID + "_" + m_graph.nodes() + "x" + m_ants.length + "x" +
              m_nIterations + "_colony.txt"));
      }
      catch (Exception e) {
        e.printStackTrace();
      }
    }

    // loop for all iterations
    while (m_nIterCounter < m_nIterations) {
      if (m_nIterCounter % 5 == 0)
        System.out.println("Iteration: " + (m_nIterCounter + 1));

      // run an iteration
      iteration();

      try {
        wait();

      }
      catch (InterruptedException ex) {
        ex.printStackTrace();

      }

      // synchronize the access to the graph
      synchronized (m_graph) {
        // apply global updating rule
        globalUpdatingRule();
      }
    }
    if (Main.GUI.makeOutput()) {
      if (m_nIterCounter == m_nIterations) {
        m_outs.close();
      }
    }
  }

  /**
   * Called for each iteration; resets the ant counter, increases the
   * iteration counter, and creates new ants (threads) for the new iteration.
   */
  private void iteration() {
    m_nAntCounter = 0;
    m_nIterCounter++;
    for (AntFramework ant : m_ants)
      ant.createThread();
  }

  /**
   * Returns the graph used by the ants
   * @return The graph
   */
  public AntGraphFramework getGraph() {
    return m_graph;
  }

  /**
   * Returns the number of ants used.
   * @return The length of the array with ant framework objects, i.e. the
   * number of ants.
   */
  public int getAnts() {
    return m_ants.length;
```

```
  }

  /**
   * The method gets the number of iterations the program should run.
   * @return The number of iterations.
   */
  public int getIterations() {
    return m_nIterations;
  }

  /**
   * The method gets the current iteration number.
   * @return The current iteration ID.
   */
  public int getIterationCounter() {
    return m_nIterCounter;
  }

  /**
   * Returns the ID of the Colony (most useful if more than one is used)
   * @return The AntColony ID.
   */
  public int getID() {
    return m_nID;
  }

  /**
   * Gets the current ant's ID.
   * @return The current value of the ant couter, i.e. the current ant's ID.
   */
  public int getAntID() {
    return m_nAntCounter;
  }

  /**
   * The update method required when implementing the Observer inteface.
   * It waits until all ants have reported back, after which it notifies
   * them, starting them on a new iteration the iteration counter hasn't
   * reached the total number of iterations.
   * @param ant The ant reporting back
   * @param obj N/A
   */
  public synchronized void update(Observable ant, Object obj) {
    m_nAntCounter++;

    if (m_nAntCounter == m_ants.length) {
      if (Main.GUI.makeOutput())
        m_outs.println(";" + AntFramework.s_dBestPathLength);
      notify();
    }
  }

  /**
   * Gets the best path distance for all the iterations.
   * @return The best tour distance.
   */
  public double getBestPathLength() {
    return AntFramework.s_dBestPathLength;
  }

  /**
   * Gets the iteration ID wehere the best tour distance was achieved.
   * @return The ID of the best iteration.
   */
  public int getLastBestPathIteration() {
    return AntFramework.s_nLastBestPathIteration;
  }

  /**
   * Checks if ants have run through all the iterations.
   * @return True if they are done, false if they need to continue.
   */
  public final boolean done() {
    return m_nIterCounter == m_nIterations;
```

```
  }

  /**
   * Abstract method for implementing the functionality for creating the ants.
   * @param graph The graph on which the ants are running.
   * @param ants The number of ants.
   * @return An array filled with objects of the type AntFramework.
   */
  protected abstract AntFramework[] createAnts(AntGraphFramework graph, int ants);

  /**
   * Abstract method global updating rule.
   */
  protected abstract void globalUpdatingRule();
}
```

E.3 AntForTSP.java

```
package tsp;

import framework.acos.*;
import java.util.*;
import Node.*;
import Node.Neighbor;

/**
 * This class is the ant for the TSP solution.
 * The parameters alfa, beta, Q0 and ksi are defined by the user in
 * the GUI. The HashMap is a map for each of the ants to see what nodes
 * that needs to be visited on the tour.
 * @author Jakob Kierkegaard & Jean-Luc Ngassa
 */
public class AntForTSP extends AntFramework {

  /**
   * The alfa value, received from the input in the GUI by the user.
   */
  private static final double A = Main.GUI.getAlfa();
  /**
   * The beta value, received from the input in the GUI by the user.
   */
  private static final double B = Main.GUI.getBeta();
  /**
   * The Q0 value, received from the input in the GUI by the user.
   */
  private static final double Q0 = Main.GUI.getQ0();
  /**
   * The ksi value, received from the input in the GUI by the user.
   */
  private static final double KSI = Main.GUI.getKsi();
  /**
   * A random number to determine whether the ant should exploit or
   * explore.
   */
  private static final Random s_randGen = new Random(7);
  /**
   * A HashMap of the nodes that the ant has not visited yet.
   */
  protected HashMap<Integer, Integer> m_nodesToVisitTble;

  public int nNode;
  public double probability, balance;

  /**
   * Creates an AntForTSP object by calling the super constructor.
   * @param startNode The ID of the node where the ant starts.
   * @param observer The observer, which is the colony.
   */
  public AntForTSP(int startNode, Observer observer) {
    super(startNode, observer);
  }
```

```java

  /**
   * Initiates the ant by calling the super method, followed by
   * initializing the HashMap, filling it with all the nodes, and
   * removing the node where the ant starts.
   */
  public void init() {
    super.init();

    final AntGraphFramework graph = s_antColony.getGraph();

    m_nodesToVisitTble = new HashMap<Integer, Integer>(graph.nodes());
    for (int i = 0; i < graph.nodes(); i++)
      m_nodesToVisitTble.put(i, i);

    m_nodesToVisitTble.remove(new Integer(m_nStartNode));
  }

  /**
   * If it is the first iteration, the first ant performs a neighborhood
   * search to deposit a basic amount of pheromone between the nodes.
   * Otherwise Q0 is dependent on a random double which dertermines if the
   * ant should expore or exploit.
   * In the end the next node is removed from the list of nodes that need
   * visiting.
   * @param nCurNode The node where the ant currently is located.
   * @return Returns the ID of the next node.
   */
  public synchronized int stateTransitionRule(int nCurNode) {
    if ((s_antColony.getIterationCounter() == 1) && (s_antColony.getAntID() == 0)) {

      final AntGraphFramework graph = s_antColony.getGraph();
      ArrayList<Node> myNodeList = graph.getNodeList();
      LinkedList<Node> myNodes = new LinkedList<Node>(myNodeList);
      myNodeList.get(nCurNode).setNeighbors(myNodes);
      for (Neighbor nabo : myNodeList.get(nCurNode).neighbors) {
        if (m_nodesToVisitTble.containsKey(nabo.toNode.nodeID)) {
          nNode = nabo.toNode.nodeID;
          break;
        }
      }

      m_nodesToVisitTble.remove(new Integer(nCurNode));
      return nNode;

    }
    else {
      final AntGraphFramework graph = s_antColony.getGraph();

      // generate a random number
      double q = s_randGen.nextDouble();
      int nMaxNode = -1;

      if (q <= Q0) {
        // Exploitation
        double dMaxVal = -1;
        double dVal;
        int nNode;

        Set keySet = m_nodesToVisitTble.keySet();

        for (Object node : keySet) {
          // select a node
          nNode = (Integer) node;

          // get the value
          dVal = graph.tau(nCurNode, nNode) *
              Math.pow(graph.etha(nCurNode, nNode), B);

          // check if it is the max
          if (dVal > dMaxVal) {
            dMaxVal = dVal;
            nMaxNode = nNode;
          }
```

```
        }
      }
      else {
        // Exploration
        double dSum = 0;
        int nNode;

        // get the sum at denominator
        Set keySet = m_nodesToVisitTble.keySet();

        for (Object node : keySet) {
          nNode = (Integer) node;

          // Update the sum
          dSum += Math.pow(graph.tau(nCurNode, nNode), A) *
                  Math.pow(graph.etha(nCurNode, nNode), B);
        }
        balance = 0;

        // search the node in agreement with eq. b)
        keySet = m_nodesToVisitTble.keySet();
        for (Object node : keySet) {
          nNode = (Integer) node;
          probability = Math.pow(graph.tau(nCurNode, nNode), A) *
                  Math.pow(graph.etha(nCurNode, nNode), B) / dSum;


          if (probability >= balance) {
            nMaxNode = nNode;
            balance = probability;
          }
        }
      }

      // delete the selected node from the list of nodes to visit
      m_nodesToVisitTble.remove(new Integer(nMaxNode));

      return nMaxNode;
    }
  }

  /**
   * The string representation of the AntForTSP object.
   * Calls the super's toString() method.
   * @return The string representation of the AntForTSP object.
   */
  public String toString() {
    String retValue;

    retValue = super.toString();
    return retValue;
  }

  /**
   * Updates the pheromone value of the edge between the current and
   * next node where the ant is located.
   * It uses formula 3.12 found in DS04 on page 78.
   * @param nCurNode The node the ant currently is at.
   * @param nNextNode The node the ant will be going to.
   */
  public void localUpdatingRule(int nCurNode, int nNextNode) {
    final AntGraphFramework graph = s_antColony.getGraph();

    double val = ((double) 1 - KSI) * graph.tau(nCurNode, nNextNode) +
                 (KSI * (graph.tau0()));

    graph.updateTau(nCurNode, nNextNode, val);
  }

  /**
   * Compares two distances, and checks which one of them is less (better)
   * @param dPathValue1 First distance.
   * @param dPathValue2 Second distance.
   * @return Returns true if the first is less, else false.
```

```
   */
  public boolean better(double dPathValue1, double dPathValue2) {
    return dPathValue1 < dPathValue2;
  }

  /**
   * Checks to see if the ant is done with the tour be checking is there
   * are more nodes in its HashMap.
   * @return True if there are no more nodes to visit.
   */
  public boolean end() {
    return m_nodesToVisitTble.isEmpty();
  }
}
```

E.4 AntFramework.java

```
package framework.acos;

import java.util.*;
import java.io.*;

/**
 * The ant framework.
 * As every ant is a thread in itself, the class implements runnable.
 * The ant is observed by the colony, meaning that the ant extends the
 * observable class.
 * @author Jakob Kierkegaard & Jean-Luc Ngassa
 */
public abstract class AntFramework extends Observable implements Runnable {
  /**
   * The ant's ID.
   */
  private final int m_nAntID;
  /**
   * A matrix to keep track of where the ant has been; when an edge is
   * visited, a 1 is entered in the appropriate spot.
   */
  protected int[][] m_path;
  /**
   * The ID of the node where the ant is right now.
   */
  protected int m_nCurNode;
  /**
   * The ID of the ant's start node.
   */
  protected final int m_nStartNode;
  /**
   * The length of the tour that has been travelled.
   */
  protected double m_dPathLength;
  /**
   * The observer of the ant (the colony).
   */
  protected final Observer m_observer;
  /**
   * The tour represented by an ArrayList of integers.
   */
  protected ArrayList<Integer> m_tour;
  /**
   * The ID counter. The current value of the counter becomes the ant's
   * ID.
   */
  private static int s_nAntIDCounter = 0;
  /**
   * The printstream if the user wants intermediate results written to
   * files.
   */
  private static PrintStream s_outs;
  /**
   * The ant's colony.
```

```
   */
  protected static AntColonyFramework s_antColony;
  /**
   * The best (shortest) length achieved so far. Set to max value so
   * improvement is always guaranteed at least once.
   */
  public static double s_dBestPathLength = Double.MAX_VALUE;
  /**
   * The best tour so far represented by an ArrayList of integers.
   */
  public static ArrayList<Integer> s_bestTour = null;
  /**
   * The iteration where the best tour was accomplished.
   */
  public static int s_nLastBestPathIteration = 0;

  /**
   * Creates a new ant.
   * @param nStartNode The ID of the starting node of the ant.
   * @param observer The observer being the ant colony.
   */
  public AntFramework(int nStartNode, Observer observer) {
    s_nAntIDCounter++;
    m_nAntID = s_nAntIDCounter;
    m_nStartNode = nStartNode;
    m_observer = observer;
  }

  /**
   * Sets the ant colony that is to be used by the ant.
   * @param antColony The ant colony that is to be used.
   */
  public static void setAntColony(AntColonyFramework antColony) {
    s_antColony = antColony;
  }

  /**
   * Resets all the variables for the ant, enabling a clean restart.
   */
  public static void reset() {
    s_dBestPathLength = Double.MAX_VALUE;
    s_bestTour = null;
    s_nLastBestPathIteration = 0;
    if (Main.GUI.makeOutput())
      s_outs = null;
  }

  /**
   * Initializes all the variables required to be an ant; the graph,
   * current node, tour list etc.
   */
  public void init() {
    if (Main.GUI.makeOutput()) {
      if (s_outs == null) {
        try {
          s_outs = new PrintStream(new FileOutputStream(Main.GUI.getOutputPath() +
            s_antColony.getID() + "_" + s_antColony.getGraph().nodes() +
            "x" + s_antColony.getAnts() + "x" + s_antColony.getIterations() +
            "_ants.txt"));
        }
        catch (Exception ex) {
          ex.printStackTrace();
        }
      }
    }

    final AntGraphFramework graph = s_antColony.getGraph();
    m_nCurNode = m_nStartNode;

    m_path = new int[graph.nodes()][graph.nodes()];
    m_tour = new ArrayList<Integer>(graph.nodes());

    m_tour.add(m_nStartNode);
    m_dPathLength = 0;
```

```
  }

  /**
   * Initializes the ant, and creates and starts a new thread.
   * The thread is based on this ant.
   */
  public void createThread() {
    init();
    new Thread(this, "Ant " + m_nAntID).start();
  }

  /**
   * The run method for the ant. The ant will be running as long as the
   * end condition isn't met. It retrieves its new node by using the
   * stateTransitionRule method, adds the new node to its tour list.
   * Afterwards it updates the pheromone on the edge (using the
   * localUpdatingRule) and calculates its tour's new distance. In the
   * end it checks if the new values are better than the old ones; if
   * this is the case, the old ones are replaced by the new ones.
   */
  public void run() {
    final AntGraphFramework graph = s_antColony.getGraph();

    // repeat while End of Activity Rule returns false
    while (!end()) {
      int nNewNode;

      // synchronize the access to the graph
      synchronized (graph) {
        // apply the State Transition Rule
        nNewNode = stateTransitionRule(m_nCurNode);

        // update the length of the path
        if (m_nCurNode != nNewNode)
          m_dPathLength += graph.delta(m_nCurNode, nNewNode);

      }
      if (m_nCurNode != nNewNode) {
        // add the current node the list of visited nodes
        m_tour.add(nNewNode);
        m_path[m_nCurNode][nNewNode] = 1;

        synchronized (graph) {
          // apply the Local Updating Rule
          localUpdatingRule(m_nCurNode, nNewNode);
        }

        // update the current node
        m_nCurNode = nNewNode;
      }
    }
    m_dPathLength += graph.delta(m_nCurNode, m_nStartNode);

    synchronized (graph) {
      // update the best tour value
      if (better(m_dPathLength, s_dBestPathLength)) {
        s_dBestPathLength = m_dPathLength;
        s_bestTour = m_tour;
        s_nLastBestPathIteration = s_antColony.getIterationCounter();

        if (Main.GUI.makeOutput())
          s_outs.println("Ant + " + m_nAntID + "," + s_dBestPathLength + "," +
                  s_nLastBestPathIteration + "," + s_bestTour.size() +
                  "," + s_bestTour);
      }
    }
    // update the observer
    m_observer.update(this, null);
    if (Main.GUI.makeOutput()) {
      if (s_antColony.done())
        s_outs.close();
    }
  }

```

```
  /**
   * Finds the better distance among the two parameters.
   * @param dPathValue The first distance to be compared.
   * @param dBestPathValue The seconds distance to be compared.
   * @return True or false.
   */
  protected abstract boolean better(double dPathValue, double dBestPathValue);

  /**
   * The abstract method defining the state transition rule. calculating
   * the next node for the ant.
   * @param r The ID of the current node of the ant.
   * @return The ID of the next node.
   */
  public abstract int stateTransitionRule(int r);

  /**
   * The abstract method handling the local updating rule, updating the
   * pheromone on the edge between node r and s.
   * @param r The ID of the node at one end of the edge.
   * @param s The ID of the node at the other end of the edge.
   */
  public abstract void localUpdatingRule(int r, int s);

  /**
   * The abstract method, defining the end condition for the ant.
   * @return True if the ant should stop.
   */
  public abstract boolean end();

  /**
   * Retrieves an Array of ints (node IDs), retrieving them from the
   * ArrayList s_bestTour.
   * @return Array of ints.
   */
  public static int[] getBestPath() {

    int nBestTourArray[] = new int[s_bestTour.size()];
    for (int i = 0; i < s_bestTour.size(); i++) {
      nBestTourArray[i] = (s_bestTour.get(i));
    }

    return nBestTourArray;
  }

  /**
   * The string representation of an ant; returning the ant's ID and
   * its current node.
   * @return The string representation of the ant.
   */
  public String toString() {
    return "Ant " + m_nAntID + ":" + m_nCurNode;
  }
}
```

E.5 AntGraphFramework.java

```
package framework.acos;

import java.util.*;
import java.io.*;

/**
 * The ant framework.
 * As every ant is a thread in itself, the class implements runnable.
 * The ant is observed by the colony, meaning that the ant extends the
 * observable class.
 * @author Jakob Kierkegaard & Jean-Luc Ngassa
 */
public abstract class AntFramework extends Observable implements Runnable {
  /**
```

```
   * The ant's ID.
   */
  private final int m_nAntID;
  /**
   * A matrix to keep track of where the ant has been; when an edge is
   * visited, a 1 is entered in the appropriate spot.
   */
  protected int[][] m_path;
  /**
   * The ID of the node where the ant is right now.
   */
  protected int m_nCurNode;
  /**
   * The ID of the ant's start node.
   */
  protected final int m_nStartNode;
  /**
   * The length of the tour that has been travelled.
   */
  protected double m_dPathLength;
  /**
   * The observer of the ant (the colony).
   */
  protected final Observer m_observer;
  /**
   * The tour represented by an ArrayList of integers.
   */
  protected ArrayList<Integer> m_tour;
  /**
   * The ID counter. The current value of the counter becomes the ant's
   * ID.
   */
  private static int s_nAntIDCounter = 0;
  /**
   * The printstream if the user wants intermediate results written to
   * files.
   */
  private static PrintStream s_outs;
  /**
   * The ant's colony.
   */
  protected static AntColonyFramework s_antColony;
  /**
   * The best (shortest) length achieved so far. Set to max value so
   * improvement is always guaranteed at least once.
   */
  public static double s_dBestPathLength = Double.MAX_VALUE;
  /**
   * The best tour so far represented by an ArrayList of integers.
   */
  public static ArrayList<Integer> s_bestTour = null;
  /**
   * The iteration where the best tour was accomplished.
   */
  public static int s_nLastBestPathIteration = 0;

  /**
   * Creates a new ant.
   * @param nStartNode The ID of the starting node of the ant.
   * @param observer The observer being the ant colony.
   */
  public AntFramework(int nStartNode, Observer observer) {
    s_nAntIDCounter++;
    m_nAntID = s_nAntIDCounter;
    m_nStartNode = nStartNode;
    m_observer = observer;
  }

  /**
   * Sets the ant colony that is to be used by the ant.
   * @param antColony The ant colony that is to be used.
   */
  public static void setAntColony(AntColonyFramework antColony) {
    s_antColony = antColony;
```

```
  }

  /**
   * Resets all the variables for the ant, enabling a clean restart.
   */
  public static void reset() {
    s_dBestPathLength = Double.MAX_VALUE;
    s_bestTour = null;
    s_nLastBestPathIteration = 0;
    if (Main.GUI.makeOutput())
      s_outs = null;
  }

  /**
   * Initializes all the variables required to be an ant; the graph,
   * current node, tour list etc.
   */
  public void init() {
    if (Main.GUI.makeOutput()) {
      if (s_outs == null) {
        try {
          s_outs = new PrintStream(new FileOutputStream(Main.GUI.getOutputPath() +
            s_antColony.getID() + "_" + s_antColony.getGraph().nodes() +
            "x" + s_antColony.getAnts() + "x" + s_antColony.getIterations() +
            "_ants.txt"));
        }
        catch (Exception ex) {
          ex.printStackTrace();
        }
      }
    }

    final AntGraphFramework graph = s_antColony.getGraph();
    m_nCurNode = m_nStartNode;

    m_path = new int[graph.nodes()][graph.nodes()];
    m_tour = new ArrayList<Integer>(graph.nodes());

    m_tour.add(m_nStartNode);
    m_dPathLength = 0;
  }

  /**
   * Initializes the ant, and creates and starts a new thread.
   * The thread is based on this ant.
   */
  public void createThread() {
    init();
    new Thread(this, "Ant " + m_nAntID).start();
  }

  /**
   * The run method for the ant. The ant will be running as long as the
   * end condition isn't met. It retrieves its new node by using the
   * stateTransitionRule method, adds the new node to its tour list.
   * Afterwards it updates the pheromone on the edge (using the
   * localUpdatingRule) and calculates its tour's new distance. In the
   * end it checks if the new values are better than the old ones; if
   * this is the case, the old ones are replaced by the new ones.
   */
  public void run() {
    final AntGraphFramework graph = s_antColony.getGraph();

    // repeat while End of Activity Rule returns false
    while (!end()) {
      int nNewNode;

      // synchronize the access to the graph
      synchronized (graph) {
        // apply the State Transition Rule
        nNewNode = stateTransitionRule(m_nCurNode);

        // update the length of the path
        if (m_nCurNode != nNewNode)
```

```
          m_dPathLength += graph.delta(m_nCurNode, nNewNode);

      }
      if (m_nCurNode != nNewNode) {
        // add the current node the list of visited nodes
        m_tour.add(nNewNode);
        m_path[m_nCurNode][nNewNode] = 1;

        synchronized (graph) {
          // apply the Local Updating Rule
          localUpdatingRule(m_nCurNode, nNewNode);
        }

        // update the current node
        m_nCurNode = nNewNode;
      }
    }
    m_dPathLength += graph.delta(m_nCurNode, m_nStartNode);

    synchronized (graph) {
      // update the best tour value
      if (better(m_dPathLength, s_dBestPathLength)) {
        s_dBestPathLength = m_dPathLength;
        s_bestTour = m_tour;
        s_nLastBestPathIteration = s_antColony.getIterationCounter();

        if (Main.GUI.makeOutput())
          s_outs.println("Ant + " + m_nAntID + "," + s_dBestPathLength + "," +
              s_nLastBestPathIteration + "," + s_bestTour.size() + "," + s_bestTour);
      }
    }
    // update the observer
    m_observer.update(this, null);
    if (Main.GUI.makeOutput()) {
      if (s_antColony.done())
        s_outs.close();
    }
  }

  /**
   * Finds the better distance among the two parameters.
   * @param dPathValue The first distance to be compared.
   * @param dBestPathValue The seconds distance to be compared.
   * @return True or false.
   */
  protected abstract boolean better(double dPathValue, double dBestPathValue);

  /**
   * The abstract method defining the state transition rule. calculating
   * the next node for the ant.
   * @param r The ID of the current node of the ant.
   * @return The ID of the next node.
   */
  public abstract int stateTransitionRule(int r);

  /**
   * The abstract method handling the local updating rule, updating the
   * pheromone on the edge between node r and s.
   * @param r The ID of the node at one end of the edge.
   * @param s The ID of the node at the other end of the edge.
   */
  public abstract void localUpdatingRule(int r, int s);

  /**
   * The abstract method, defining the end condition for the ant.
   * @return True if the ant should stop.
   */
  public abstract boolean end();

  /**
   * Retrieves an Array of ints (node IDs), retrieving them from the
   * ArrayList s_bestTour.
   * @return Array of ints.
   */
```

```
  public static int[] getBestPath() {

    int nBestTourArray[] = new int[s_bestTour.size()];
    for (int i = 0; i < s_bestTour.size(); i++) {
      nBestTourArray[i] = (s_bestTour.get(i));
    }

    return nBestTourArray;
  }

  /**
   * The string representation of an ant; returning the ant's ID and
   * its current node.
   * @return The string representation of the ant.
   */
  public String toString() {
    return "Ant " + m_nAntID + ":" + m_nCurNode;
  }
}
```

E.6 GUI.java

```
package Main;

import IO.*;
import Node.Node;
import framework.acos.*;
import tsp.*;
import tsp.optimization.*;
import java.util.*;
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;

/**
 * The GUI class, which retrieves the parameters alfa, beta, rho, ksi,
 * W and Q0 from the user. It also lets the user choose whether or not
 * he/she wants preliminary results printed to files in a folder chosen
 * by the user, and the instance (.tsp file) on which the program should
 * use.
 * @author Jakob Kierkegaard & Jean-Luc Ngassa
 */
public class GUI extends JFrame implements ActionListener {

  private static AntColonyForTSP antColony;
  public static TSPCanvas canvas = new TSPCanvas();
  private static JTextField filePath, outputPath, nuOfAnts, nuOfIterations, result;
  private Button runButton, defaultButton, clearButton, locateTspFileButton;
  private Button locateOutputFolderButton;
  private static JCheckBox giveOutput;
  private static JCheckBox doTwoOpt, doTwoHOpt, doOpt;
  private JPanel canvasJPanel;
  private static JSpinner alfaSpinner, betaSpinner, wSpinner, q0Spinner;
  private static JSpinner ksiSpinner, rhoSpinner;
  private static JLabel statusJLabel, tourLengthJLabel, totalTimeJLabel;

  private static final String ALFA = "\u03B1";
  private static final String BETA = "\u03B2";
  private static final String RHO = "\u03C1";
  private static final String KSI = "\u03BE";
  private static final int WINDOW_WIDTH = 1000;
  private static final int WINDOW_HEIGHT = 800;
  public static final int CANVAS_WIDTH = WINDOW_WIDTH / 2 - 40;
  public static final int CANVAS_HEIGHT = WINDOW_HEIGHT - 200;

  /**
   * The constructor calls the displayGUI method.
   */
  public GUI() {
```

```java
    displayGUI();
  }

  /**
   * The displayGUI method creates and displays the entire GUI. It
   * also adds actionListeners where required (buttons and checkboxes).
   */
  private void displayGUI() {
    JPanel parameterJPanel, buttonJPanel, numberJPanel, optimizationJPanel;
    JPanel fileJPanel, extraJPanel, statusJPanel, visualJPanel, dataJPanel;
    JPanel innerPJPanel;
    SpinnerModel alfaSpinnerModel, betaSpinnerModel, wSpinnerModel;
    SpinnerModel rhoSpinnerModel, ksiSpinnerModel, q0SpinnerModel;
    JTextArea resultArea;
    JScrollPane scrollPane;

    setSize(new Dimension(WINDOW_WIDTH, WINDOW_HEIGHT));
    setTitle("TSP output");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setLayout(new GridLayout(1, 2));

    // The canvas JPanel
    visualJPanel = new JPanel();
    visualJPanel.setLayout(new BorderLayout());

    canvasJPanel = new JPanel(new BorderLayout());
    canvasJPanel.setBorder(BorderFactory.createBevelBorder(BevelBorder.RAISED));
    canvasJPanel.add(new Label("Visual representation of the TSP",
                    Label.CENTER), "North");

    dataJPanel = new JPanel();
    dataJPanel.setLayout(new GridLayout(2, 1));
    dataJPanel.setBorder(BorderFactory.createBevelBorder(BevelBorder.RAISED));

    tourLengthJLabel = new JLabel();
    result = new JTextField(40);
    resultArea = new JTextArea(1, 40);
    resultArea.setEditable(false);
    result.setEditable(false);

    scrollPane = new JScrollPane();
    scrollPane.setVerticalScrollBarPolicy(
                    ScrollPaneConstants.VERTICAL_SCROLLBAR_NEVER);
    scrollPane.setHorizontalScrollBarPolicy(
                    ScrollPaneConstants.HORIZONTAL_SCROLLBAR_ALWAYS);
    scrollPane.setColumnHeaderView(result);

    dataJPanel.add(tourLengthJLabel);
    dataJPanel.add(scrollPane);

    visualJPanel.add(canvasJPanel);
    visualJPanel.add(dataJPanel, "South");

    // The parameter JPanel
    parameterJPanel = new JPanel(new BorderLayout());
    parameterJPanel.setBorder(BorderFactory.createBevelBorder(BevelBorder.RAISED));
    innerPJPanel = new JPanel(new GridLayout(3, 1));
    parameterJPanel.add(new Label("Parameters", Label.CENTER), "North");

    // The button JPanel
    buttonJPanel = new JPanel(new FlowLayout());
    buttonJPanel.setBorder(BorderFactory.createEtchedBorder());
    runButton = new Button("RUN");
    clearButton = new Button("Clear settings");
    defaultButton = new Button("Use default values");

    runButton.addActionListener(this);
    clearButton.addActionListener(this);
    defaultButton.addActionListener(this);

    buttonJPanel.add(runButton);
    buttonJPanel.add(clearButton);
    buttonJPanel.add(defaultButton);

```

```
    // The number JPanel
    numberJPanel = new JPanel(new GridLayout(9, 2));

    alfaSpinnerModel = new SpinnerNumberModel(0.1, 0.0, 1.0, 0.01);
    betaSpinnerModel = new SpinnerNumberModel(2.0, 1.0, 10.0, 0.1);
    wSpinnerModel = new SpinnerNumberModel(1, 1, 100, 1);
    q0SpinnerModel = new SpinnerNumberModel(0.8, 0.0, 1.0, 0.01);
    rhoSpinnerModel = new SpinnerNumberModel(0.1, 0.0, 1.0, 0.01);
    ksiSpinnerModel = new SpinnerNumberModel(0.1, 0.0, 1.0, 0.01);

    nuOfAnts = new JTextField(20);
    nuOfIterations = new JTextField(20);
    nuOfAnts.setSize(100, 10);
    nuOfIterations.setSize(100, 10);

    alfaSpinner = addSpinner(numberJPanel, ALFA, alfaSpinnerModel);
    betaSpinner = addSpinner(numberJPanel, BETA, betaSpinnerModel);
    wSpinner = addSpinner(numberJPanel, "W", wSpinnerModel);
    q0Spinner = addSpinner(numberJPanel, "Q0", q0SpinnerModel);
    rhoSpinner = addSpinner(numberJPanel, RHO, rhoSpinnerModel);
    ksiSpinner = addSpinner(numberJPanel, KSI, ksiSpinnerModel);

    numberJPanel.add(new Label("Number of ants"));
    numberJPanel.add(nuOfAnts);
    numberJPanel.add(new Label("Number of iterations"));
    numberJPanel.add(nuOfIterations);
    numberJPanel.setBorder(BorderFactory.createTitledBorder("Numbers"));

    /**
     * This internal class extends the existing InputVerifier,
     * overwriting the constructor. Used to limit input in the
     * textfields to only be positive integers.
     */
    class intVerifier extends InputVerifier {
      public final boolean verify(JComponent comp) {
        boolean returnValue = false;
        JTextField textField = (JTextField) comp;
        try {
          if (Integer.parseInt(textField.getText()) > 0)
            returnValue = true;
          else
            Toolkit.getDefaultToolkit().beep();
        }
        catch (NumberFormatException e) {
          Toolkit.getDefaultToolkit().beep();
        }
        return returnValue;
      }
    }

    nuOfAnts.setInputVerifier(new intVerifier());
    nuOfIterations.setInputVerifier(new intVerifier());

    // The file choosers
    fileJPanel = new JPanel(new FlowLayout());
    fileJPanel.setBorder(BorderFactory.createTitledBorder("File settings"));
    locateTspFileButton = new Button("Locate TSP file");

    locateTspFileButton.addActionListener(this);

    filePath = new JTextField(40);
    fileJPanel.add(locateTspFileButton);
    fileJPanel.add(filePath);

    giveOutput = new JCheckBox("Do you want intermediate results printed to files?",
              false);
    locateOutputFolderButton = new Button("Find output folder");
    locateOutputFolderButton.setEnabled(false);
    outputPath = new JTextField(40);
    outputPath.setEnabled(false);

    giveOutput.addActionListener(this);
    locateOutputFolderButton.addActionListener(this);

```

```
    fileJPanel.add(giveOutput);
    fileJPanel.add(locateOutputFolderButton);
    fileJPanel.add(outputPath);

    // The optimization choices
    optimizationJPanel = new JPanel(new GridLayout(2, 2));
    optimizationJPanel.setBorder(BorderFactory.createTitledBorder("Optimization"));
    doOpt = new JCheckBox("Use local search optimization", true);
    doTwoOpt = new JCheckBox("Use 2-opt", true);
    doTwoHOpt = new JCheckBox("Use 2 -opt", true);
    doOpt.addActionListener(this);

    optimizationJPanel.add(doOpt);
    optimizationJPanel.add(new JLabel());
    optimizationJPanel.add(doTwoOpt);
    optimizationJPanel.add(doTwoHOpt);

    // The status panel
    statusJPanel = new JPanel(new GridLayout(2, 1));
    statusJPanel.setBorder(BorderFactory.createTitledBorder("Status"));
    statusJLabel = new JLabel("", SwingConstants.CENTER);
    statusJLabel.setVisible(true);
    totalTimeJLabel = new JLabel("", SwingConstants.CENTER);
    totalTimeJLabel.setVisible(true);

    statusJPanel.add(statusJLabel);
    statusJPanel.add(totalTimeJLabel);

    // Adding everything
    extraJPanel = new JPanel(new GridLayout(2, 1));
    extraJPanel.add(optimizationJPanel);
    extraJPanel.add(statusJPanel);

    innerPJPanel.add(numberJPanel);
    innerPJPanel.add(fileJPanel);
    innerPJPanel.add(extraJPanel);
    parameterJPanel.add(buttonJPanel, "South");
    parameterJPanel.add(innerPJPanel);

    this.add(parameterJPanel);
    this.add(visualJPanel);
    this.setVisible(true);
    canvas.repaint();
  }

  /**
   * The actionperformed method handles all actions when the user
   * interacts with the GUI.
   * @param e The event.
   */
  public void actionPerformed(ActionEvent e) {
    JFileChooser fc;
    String missingParameters;
    double startTime, endTime, totalTime;

    if (e.getSource() == doOpt) {
      if (doOpt.isSelected()) {
        doTwoOpt.setEnabled(true);
        doTwoHOpt.setEnabled(true);
      }
      else {
        doTwoOpt.setEnabled(false);
        doTwoHOpt.setEnabled(false);
      }
    }
    else if (e.getSource() == giveOutput) {
      if (giveOutput.isSelected()) {
        locateOutputFolderButton.setEnabled(true);
        outputPath.setEnabled(true);
      }
      else {
        locateOutputFolderButton.setEnabled(false);
        outputPath.setEnabled(false);
      }
```

```java
    }
    else if (e.getSource() == runButton) {
      missingParameters = "";
      /**
       * Checks if all the required fields have gotten values.
       * if this isn't the case, a warning message comes up to inform
       * the user about what is missing.
       */
      if (nuOfAnts.getText().equals("") ||
          nuOfIterations.getText().equals("") ||
          filePath.getText().equals("") ||
          (giveOutput.isSelected() && outputPath.getText().equals("")) ||
          (doOpt.isSelected() && (!doTwoOpt.isSelected() &&
                      !doTwoHOpt.isSelected()))) {
        if (nuOfAnts.getText().equals(""))
          missingParameters += "Number of ants\n";
        if (nuOfIterations.getText().equals(""))
          missingParameters += "Number of iterations\n";
        if (filePath.getText().equals(""))
          missingParameters += "The tsp file\n";
        if (giveOutput.isSelected() && outputPath.getText().equals(""))
          missingParameters += "The path where you want your preliminary
                        output\n";
        if (doOpt.isSelected() && (!doTwoOpt.isSelected() ||
                      !doTwoHOpt.isSelected()))
          missingParameters += "The type of local search optimization\n";
        JOptionPane.showMessageDialog(this,
            "Please fill out all the parameters.\n\nYou are missing the
              following:\n" +
            missingParameters, "Warning", JOptionPane.WARNING_MESSAGE);
      }
      else {
        /**
         * If the user wants intermediate results printed to files,
         * he/she is warned that the program will require much
         * more time to run. If the user accepts, the time is
         * registered and the runProgram method is called. When
         * done, the time is registered again, and the total run
         * time is found and printed.
         */
        int n = JOptionPane.YES_OPTION;
        if (giveOutput.isSelected()) {
          n = JOptionPane.showConfirmDialog(this,
              "It will take an extended amount of time\nto print the
              results to files.\n\n
              Do you want to continue?", "Warning", JOptionPane.YES_NO_OPTION,
              JOptionPane.WARNING_MESSAGE);
        }
        if (n == JOptionPane.YES_OPTION) {
          statusJLabel.setText("Working...");
          totalTimeJLabel.setText("");

          startTime = System.currentTimeMillis();

          runProgram();

          endTime = System.currentTimeMillis();
          totalTime = endTime - startTime;

          statusJLabel.setText("Done!");
          statusJLabel.repaint();

          totalTimeJLabel.setText("It took " + totalTime / 1000 +
                      " seconds to compute the result");
          totalTimeJLabel.repaint();

          canvasJPanel.add(canvas, "Center");
          canvas.repaint();
        }
      }
    }
    else if (e.getSource() == clearButton) {
      nuOfAnts.setText("");
      nuOfIterations.setText("");
```

```
      filePath.setText("");
      outputPath.setText("");
      giveOutput.setSelected(false);
      locateOutputFolderButton.setEnabled(false);
      outputPath.setEnabled(false);
      alfaSpinner.setValue(0);
      betaSpinner.setValue(1);
      wSpinner.setValue(1);
      q0Spinner.setValue(0);
      rhoSpinner.setValue(0);
      doOpt.setSelected(false);
      doTwoOpt.setSelected(false);
      doTwoHOpt.setSelected(false);
      doTwoOpt.setEnabled(false);
      doTwoHOpt.setEnabled(false);
    }
    else if (e.getSource() == defaultButton) {
      nuOfAnts.setText("51");
      nuOfIterations.setText("100");
      outputPath.setText("");
      giveOutput.setSelected(false);
      locateOutputFolderButton.setEnabled(false);
      outputPath.setEnabled(false);
      alfaSpinner.setValue(0.1);
      betaSpinner.setValue(2);
      wSpinner.setValue(1);
      q0Spinner.setValue(0.8);
      rhoSpinner.setValue(0.1);
      doOpt.setSelected(true);
      doTwoOpt.setSelected(true);
      doTwoHOpt.setSelected(true);
      doTwoOpt.setEnabled(true);
      doTwoHOpt.setEnabled(true);
    }
    else if (e.getSource() == locateTspFileButton) {
      fc = new JFileChooser();

      TSPFileFilter filter = new TSPFileFilter();
      fc.setFileFilter(filter);

      int returnValue = fc.showOpenDialog(this);
      if (returnValue == JFileChooser.APPROVE_OPTION) {
        File file = fc.getSelectedFile();
        filePath.setText(file.getAbsoluteFile().toString());
      }
    }
    else if (e.getSource() == locateOutputFolderButton) {
      fc = new JFileChooser();
      fc.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);

      int returnValue = fc.showOpenDialog(this);
      if (returnValue == JFileChooser.APPROVE_OPTION) {
        File file = fc.getSelectedFile();
        outputPath.setText(file.getAbsoluteFile().toString());
      }
    }
  }

  /**
   * A small method to simplify adding the spinners to the GUI, as it
   * can be a fairly complex operation. It adds the spinners together
   * with a small associated string to explain what value is set using
   * that specific spinner.
   * @param c The container which is to hold the spinner.
   * @param label The label that holds the explanitory text.
   * @param model The model of the spinner.
   * @return The JSpinner with the applied model.
   */
  private JSpinner addSpinner(Container c, String label, SpinnerModel model) {
    JLabel l = new JLabel(label);
    c.add(l);
    JSpinner spinner = new JSpinner(model);
    ((JSpinner.DefaultEditor) spinner.getEditor()).getTextField().setColumns(10);
    l.setLabelFor(spinner);
```

```
    c.add(spinner);
    return spinner;
  }

  /**
   * The method that starts the calculation of the best tour.
   * It retrieves an InputFile, from which it acquires the nodes and
   * creates the delta matrix based on these nodes and their distances
   * to eachother. A new graph is created using this matrix, and the
   * graph is used for creating a new colony.
   * Optimization is done accordingly to the user's wishes, and the
   * final result is printed to the GUI in the respective places.
   */
  private void runProgram() {
    int nNodes, nAnts, nIterations;
    ArrayList<Node> nodes;
    TwoOpt twoOpt;
    TwohOpt twohOpt;

    try {
      new InputFile(filePath.getText());
    }
    catch (Exception e) {
      System.out.println("File error: " + e);
    }

    nodes = InputFile.getNode();
    nNodes = InputFile.getNode().size();
    nAnts = Integer.parseInt(nuOfAnts.getText());
    nIterations = Integer.parseInt(nuOfIterations.getText());

    double delta[][] = new double[nNodes][nNodes];
    for (int i = 0; i < nNodes; i++) {
      for (int j = i + 1; j < nNodes; j++) {
        double ix, iy, jx, jy;
        ix = nodes.get(i).x;
        iy = nodes.get(i).y;
        jx = nodes.get(j).x;
        jy = nodes.get(j).y;

        delta[j][i] = delta[i][j] = distance(ix, iy, jx, jy);
      }
    }
    AntGraphFramework graph = new AntGraphFramework(nodes, delta);

    try {
      if (makeOutput()) {
        ObjectOutputStream outs = new ObjectOutputStream(new
            FileOutputStream(getOutputPath() + nNodes + "_antgraph.bin"));
        outs.writeObject(graph.toString());
        outs.close();


        FileOutputStream outs1 = new FileOutputStream(getOutputPath() +
                              nNodes + "_antgraph.txt");

        for (int i = 0; i < nNodes; i++) {
          for (int j = 0; j < nNodes; j++) {
            outs1.write((graph.delta(i, j) + ",").getBytes());
          }
          outs1.write('\n');
        }
        outs1.close();
      }

      if (makeOutput()) {
        PrintStream outs2 = new PrintStream(new FileOutputStream(getOutputPath() +
                  nNodes + "x" + nAnts + "x" + nIterations + "_results.txt"));
        graph.resetTau();
        antColony = new AntColonyForTSP(graph, nAnts, nIterations);
        antColony.start();

        outs2.println(1 + "," + antColony.getBestPathLength() + "," +
                antColony.getLastBestPathIteration());
```

```
          outs2.close();
        }
        else {
          graph.resetTau();
          antColony = new AntColonyForTSP(graph, nAnts, nIterations);
          antColony.start();
        }
      }
      catch (Exception e) {
        System.out.println(e + "no file output");
      }

      canvasJPanel.remove(canvas);
      if (!doOpt.isSelected()) {
        tourLengthJLabel.setText(String.valueOf(antColony.getBestPathLength()));
        result.setText(antColony.getBestTour().toString());
        canvas = new TSPCanvas(antColony.getBestTourNodes());
      }
      else {
        if (!doTwoHOpt.isSelected()) {
          twoOpt = new TwoOpt(antColony.nodesTour(antColony.getBestTour(),
                                antColony.getBestPathLength());
          tourLengthJLabel.setText(String.valueOf(twoOpt.bestLength));
          result.setText(twoOpt.printTour(twoOpt.nodeTour));
          canvas = new TSPCanvas(twoOpt);
        }
        else {
          if (!doTwoOpt.isSelected()) {
            twohOpt = new TwohOpt(antColony.nodesTour(antColony.getBestTour(),
                                  antColony.getBestPathLength());
          }
          else {
            twoOpt = new TwoOpt(antColony.nodesTour(antColony.getBestTour(),
                                antColony.getBestPathLength());
            twohOpt = new TwohOpt(twoOpt.nodeTour, twoOpt.bestLength);
          }
          tourLengthJLabel.setText(String.valueOf(twohOpt.bestLength));
          result.setText(twohOpt.printTour(twohOpt.nodeTour));
          canvas = new TSPCanvas(twohOpt);
        }
      }
    }

    /**
     * Calculates the distance between two nodes using pytagoras.
     * @param x X-coordinate of the first node.
     * @param y Y-coordinate of the first node.
     * @param a X-coordinate of the second node.
     * @param b Y-coordinate of the second node.
     * @return The distance as an int.
     */
    private static int distance(double x, double y, double a, double b) {
      return (int) (Math.sqrt(Math.pow(x - a, 2) + Math.pow(y - b, 2)) + 0.5);
    }

    /**
     * The main method, calling the constructor.
     * @param args Arguments when running the program, not used.
     */
    public static void main(String[] args) {
      new GUI();
    }

    /**
     * Retrieves the value set in the beta spinner.
     * @return  The beta value.
     */
    public static double getBeta() {
      return Double.parseDouble(betaSpinner.getValue().toString());
    }

    /**
     * Retrieves the value set in the rho spinner.
     * @return The rho value.
```

```
   */
  public static double getRho() {
    return Double.parseDouble(rhoSpinner.getValue().toString());
  }

  /**
   * Retrieves the value set in the Q0 spinner.
   * @return The Q0 value.
   */
  public static double getQ0() {
    return Double.parseDouble(q0Spinner.getValue().toString());
  }

  /**
   * Retrieves the value set in the W spinner.
   * @return The W value.
   */
  public static int getW() {
    return Integer.parseInt(wSpinner.getValue().toString());
  }

  /**
   * Retrieves the value set in the alfa spinner.
   * @return The alfa value.
   */
  public static double getAlfa() {
    return Double.parseDouble(alfaSpinner.getValue().toString());
  }

  /**
   * Retrieves the value set in the ksi spinner.
   * @return The ksi value.
   */
  public static double getKsi() {
    return Double.parseDouble(ksiSpinner.getValue().toString());
  }

  /**
   * Checks if the user has accepted to get intermediate results printed
   * to files defined in the output path.
   * @return True if the user wants output, else false.
   */
  public static boolean makeOutput() {
    return giveOutput.isSelected();
  }

  /**
   * Gets the path to where the user wants his files with the
   * intermediate results written.
   * @return The chosen path.
   */
  public static String getOutputPath() {
    return outputPath.getText() + "\\";
  }
}
```

E.7 InputFile.java

```
package IO;

import Node.Node;
import java.io.*;
import java.util.*;

/**
 * This class retrives an instance file, and converts the data in the
 * file to a manageble ArrayList of nodes.
 * @author Jakob Kierkegaard & Jean-Luc Ngassa
 */
public class InputFile {
  private static ArrayList<Node> nodeList = new ArrayList<Node>();
```

```

  /**
   * The constructor takes a string with the name and filename of the
   * file containing the instance, creates nodes based on the data
   * retrived, and puts the nodes into an ArrayList.
   * @param file The path and filename of the file containg the instance.
   */
  public InputFile(String file) {
    Scanner scan;
    BufferedReader in;
    String inputLine;
    int nodeId;
    double x, y;

    nodeList.clear();
    try {
      in = new BufferedReader(new FileReader(file));
      inputLine = in.readLine();
      while (!inputLine.equalsIgnoreCase("NODE_COORD_SECTION")) {
        inputLine = in.readLine();
      }
      inputLine = in.readLine();
      while (!inputLine.equalsIgnoreCase("EOF")) {
        scan = new Scanner(inputLine);
        nodeId = scan.nextInt() - 1;
        x = scan.nextDouble();
        y = scan.nextDouble();
        nodeList.add(new Node(nodeId, x, y));
        inputLine = in.readLine();
      }
    }
    catch (IOException e) {
      System.out.println("File not found" + e);
    }
  }

  /**
   * Retrieves the ArrayList created using the instance file.
   * @return ArrayList of node objects.
   */
  public static ArrayList<Node> getNode() {
    return nodeList;
  }
}
```

E.8 Neighbor.java

```
package Node;

/**
 * A neighbor is a node with that node's distance to the node which a
 * neighborlist belongs. Implements comparable so that the  neighbors
 * can be sorted by distance.
 * @author Jakob Kierkegaard & Jean-Luc Ngassa
 */
public class Neighbor implements Comparable<Neighbor> {

  public final Node toNode;
  public final double distance;

  /**
   * Creates a new distance using a node and distance.
   * @param toNode The node that has the distance to the node to which
   * the neighbor belongs.
   * @param distance The distance to the node from the node to which the
   * neighbor belongs.
   */
  public Neighbor(Node toNode, double distance) {
    this.toNode = toNode;
    this.distance = distance;
  }
```

```

  /**
   * The implemented compareTo method.
   * @param other Another neighbor.
   * @return Returns 1 if this neighbor is further away, 0 if it is
   * closer or 0 if they are at the same distance to the node that has
   * them as neighbors.
   */
  public int compareTo(Neighbor other) {
    double diff = distance - other.distance;
    return diff > 0 ? 1 : diff == 0 ? 0 : -1;
  }

  /**
   * Returns a string representation of the neighbor.
   * @return The string representation of the neighbor.
   */
  public String toString() {
    return "Node ID: " + (toNode.nodeID) + "\tDistance: " + distance;
  }
}
```

E.9 Node.java

```
package Node;

import java.util.*;

/**
 * The Node class handles the nodes in the program.
 * A node contains an ID, x and y coordinates, and a priority queue of
 * its neighbors.
 * @author Jakob Kierkegaard & Jean-Luc Ngassa
 */
public class Node {

  public final int nodeID;
  public final double x;
  public final double y;
  public PriorityQueue<Neighbor> neighbors = new PriorityQueue<Neighbor>();

  /**
   * The constructor creates a new node based on the ID and coordinates
   * found in the parameters.
   * @param nodeID The node's ID.
   * @param x The node's x coordinate.
   * @param y The node's y coordinate.
   */
  public Node(int nodeID, double x, double y) {
    this.nodeID = nodeID;
    this.x = x;
    this.y = y;
  }

  /**
   * Calculates the distance between this node and another node using
   * pytagorass
   * @param other The node to which the distance is calculated.
   * @return A distance of the type int between the two nodes.
   */
  public int distance(Node other) {
    return (int) (Math.sqrt(Math.pow(this.x - other.x, 2) +
            Math.pow(this.y - other.y, 2)) + 0.5);
  }

  /**
   * Creates a priority queue of neighbors, sorting them by distance from
   * low to high. Puts in all of the nodes from the parameter into the
   * neighbor list except for itself.
   * @param theNodes The list of nodes in the instance.
   */
```

```
  public void setNeighbors(LinkedList<Node> theNodes) {
    ArrayList<Neighbor> nodesList = new ArrayList<Neighbor>();
    for (Node node : theNodes) {
      if (!node.equals(this))
        nodesList.add(new Neighbor(node, distance(node)));
    }
    Collections.sort(nodesList);
    neighbors = new PriorityQueue<Neighbor>(nodesList);
  }

  /**
   * Returns a string representation of the node.
   * @return The string representation of the node.
   */
  public String toString() {
    return nodeID + ", x: " + x + ", y: " + y;
  }
}
```

E.10 TSPCanvas.java

```
package Main;

import java.awt.*;
import java.util.*;
import tsp.optimization.*;
import Node.Node;

/**
 * This is our personalized version of a canvas, which implements
 * functionality that enables it to print a tour; edges and nodes.
 * @author Jakob Kierkegaard & Jean-Luc Ngassa
 */
public class TSPCanvas extends Canvas {

  private LinkedList<Node> tour;

  private int length = 0;
  private int greatestX, greatestY, smallestX, smallestY;
  private double scale = 0.0;
  private int[] xCoords;
  private int[] yCoords;

  /**
   * Creates an empty TSPCanvas with a white background.
   */
  public TSPCanvas() {
    this.setBackground(Color.WHITE);
  }

  /**
   * Creates a TSPCanvas based on a LinkedList of nodes which represents
   * the tour. The size is dependent on the size of the window size defined
   * in the GUI.
   * @param t The tour as a LinkedList
   */
  public TSPCanvas(LinkedList<Node> t) {
    this.setBackground(Color.WHITE);
    this.tour = t;
    this.setSize(GUI.CANVAS_WIDTH, GUI.CANVAS_HEIGHT);
    length = tour.size();
    scaleTour();
  }

  /**
   * The constructor takes a TwoOpt object, and takes its tour object
   * and passes it on to the class' other constructor.
   * @param t The TwoOpt object.
   */
  public TSPCanvas(TwoOpt t) {
    this(t.nodeTour);
```

```
  }

  /**
   * The constructor takes a TwohOpt object, and takes its tour object
   * and passes it on to the class' other constructor.
   * @param t The TwohOpt object.
   */
  public TSPCanvas(TwohOpt t) {
    this(t.nodeTour);
  }

  /**
   * The paint method of the canvas. It paints the nodes as small red
   * circles, and the edges are black lines between them.
   * @param g The Graphics object.
   */
  public void paint(Graphics g) {
    g.setColor(Color.RED);
    for (int i = 0; i < length; i++)
      g.drawOval(xCoords[i] - 2, yCoords[i] - 2, 4, 4);
    g.setColor(Color.BLACK);
    g.drawPolyline(xCoords, yCoords, length);
    g.drawLine(xCoords[0], yCoords[0], xCoords[length - 1], yCoords[length - 1]);
  }

  /**
   * To make sure that the graphical representation of the tour fills
   * out the canvas' space (and not just sits in a corner), we regulate
   * the nodes' coordinates, so that we enlarge or shrink the tour so
   * that it fits to all sides.
   * We do not stretch the tour.
   */
  private void scaleTour() {
    xCoords = new int[length];
    yCoords = new int[length];

    for (int i = 0; i < length; i++) {
      xCoords[i] = (int) (tour.get(i).x);
      if (i == 0)
        greatestX = smallestX = xCoords[i];
      if (xCoords[i] > greatestX)
        greatestX = xCoords[i];
      if (xCoords[i] < smallestX)
        smallestX = xCoords[i];
      yCoords[i] = (int) (tour.get(i).y);
      if (i == 0)
        greatestY = smallestY = yCoords[i];
      if (yCoords[i] > greatestY)
        greatestY = yCoords[i];
      if (yCoords[i] < smallestY)
        smallestY = yCoords[i];
    }
    adjustToCoordinates();

    double scaleX = (double) GUI.CANVAS_WIDTH / (double) greatestX;
    double scaleY = (double) GUI.CANVAS_HEIGHT / (double) greatestY;

    scale = Math.min(scaleX, scaleY);

    adjustToScale();
  }

  /**
   * The scale between the largest x-coordinate and the width of the
   * canvas or between the largest y-coordinate and the height of the
   * canvas (depending on which is the smallest) is applied to the nodes'
   * coordinates, enlarging/shrinking the tour so it fits.
   */
  private void adjustToScale() {
    for (int i = 0; i < length; i++) {
      xCoords[i] = (int) ((xCoords[i] * scale) + 0.5);
      yCoords[i] = (int) ((yCoords[i] * scale) + 0.5);
    }
  }
```

```

  /**
   * Because our canvas' lowest value in the coordinate system is 0, we
   * have to regulate the coordinates, if there are any below 0. This is
   * simply done by adding the lowest X to all the x-coordinates and
   * similar with the y coordinates.
   * 5 is added to the coordinates so that there is a little free space
   * between the graphical representation and the canvas' border.
   */
  private void adjustToCoordinates() {
    greatestX = greatestX - smallestX + 5;
    greatestY = greatestY - smallestY + 5;
    for (int i = 0; i < length; i++) {
      xCoords[i] = xCoords[i] - smallestX + 5;
      yCoords[i] = yCoords[i] - smallestY + 5;
    }
  }
}
```

E.11 TSPFileFilter.java

```
/**
 * This class is heavily inspired by FileChooserDemo2.java
 * and ExampleFileFilter.java that can be found on Sun's homepage.
 */

package Main;

import java.io.File;
import javax.swing.filechooser.*;

/**
 * This class creates a file filter, which enables us to limit what
 * file types ae to be seen when the user wants to find a .tsp file
 * when running the program.
 * @author Jakob Kierkegaard & Jean-Luc Ngassa
 */
public final class TSPFileFilter extends FileFilter {

  /**
   * The method gets the extension of the file in question to check if
   * it should be visible in the file chooser which applies the file
   * filter.
   * @param file The path and filename of the file in question.
   * @return Returns the extension of the file as a string.
   */
  private static String getExtension(File file) {
    if (file != null) {
      String filename = file.getName();
      int i = filename.lastIndexOf(".");
      if (i > 0 && i < filename.length() - 1)
        return filename.substring(i + 1).toLowerCase();
    }
    return null;
  }

  /**
   * Checks if the file passed as a parameter should be visible in the
   * file chooser. It also lets folders in the file chooser be visible.
   * @param file The path and filename of the file in question.
   * @return Returns true if it should be visible in the file chooser.
   */
  public final boolean accept(File file) {
    if (file.isDirectory())
      return true;
    if (getExtension(file) != null) {
      if (getExtension(file).equals("tsp"))
        return true;
    }
    return false;
  }
```

```

  /**
   * Gets the description of what files are visible.
   * @return The description of what files are visible.
   */
  public final String getDescription() {
    return "TSP files";
  }
}
```

E.12 TwohOpt.java

```
package tsp.optimization;

import java.util.*;

import Node.*;

/**
 * This class takes a tour and does a 2-opt local search on it.
 * @author Jakob Kierkegaard & Jean-Luc Ngassa
 */
public class TwohOpt {

  public double bestLength;
  private Node t1, t3;
  public LinkedList<Node> nodeTour = new LinkedList<Node>();

  /**
   * The constructor instantiates the distance of the tour and the tour
   * itself.
   * @param tour The tour that needs optimizing.
   * @param bestLength The distance of the tour that needs optimizing.
   */
  public TwohOpt(LinkedList<Node> tour, double bestLength) {
    this.bestLength = bestLength;
    this.nodeTour = tour;
    twohOpt();
  }

  /**
   * Gets the node before a specific node. If the specific node is in
   * the bottom of the list, it gets the last item in the list, else it
   * gets the node at the index before.
   * @param curNode The node to which the found previous node belongs.
   * @return The previous node.
   */
  private Node pred(Node curNode) {
    if (nodeTour.indexOf(curNode) == 0)
      return nodeTour.getLast();
    else {
      return nodeTour.get(nodeTour.indexOf(curNode) - 1);
    }
  }

  /**
   * Gets the node after a specific node. Gets the node in the index
   * after, and the modulo takes care of the scenario where the
   * specified node is in the end of the list.
   * @param curNode The node to which the found succeeding node belongs.
   * @return The succeeding node.
   */
  private Node suc(Node curNode) {
    return nodeTour.get((nodeTour.indexOf(curNode) + 1) % nodeTour.size());
  }

  /**
   * The swap is simply the movement of a node (T3) from any place in
   * the tour an in between T1 and T2. When using a LinkedList as we are,
   * this is simply done by removing it from one place, and inserting it
   * into the index after the index of T1.
```

```
   */
  private void swap() {
    nodeTour.remove(t3);
    nodeTour.add(nodeTour.indexOf(t1) + 1, t3);
  }

  /**
   * Gives a string representation of the tour.
   * @param tour The tour as a LinkedList.
   * @return The string representation of the tour.
   */
  public String printTour(LinkedList<Node> tour) {
    String output = "[";
    for (Node n : tour) {
      output += " " + n.nodeID + ",";
    }
    output += "]";
    return output;
  }

  /**
   * The method goes through the entire tour to look for any feasible
   * swaps. It gets a node T1, and looks through its neighbors to see
   * if there are any advantagous swaps.
   * As the tour changes after each swap, we let it start over every
   * time a swap is made to make sure we get all possible swaps.
   */
  private void twohOpt() {
    double gain;
    boolean improved;
    Node t2, t4, t5;

    Node firstNode = t1 = nodeTour.get(0);
    do
      t1.setNeighbors(nodeTour);
    while ((t1 = suc(t1)) != firstNode);
    do {
      improved = false;
      t1 = firstNode;
      do {
        t2 = suc(t1);
        for (Neighbor n : t1.neighbors) {
          t3 = n.toNode;
          if (t3 != t2) {
            t4 = pred(t3);
            t5 = suc(t3);
            if ((t4 != t1) && (t5 != t2) && (t1 != t5) && (t2 != t4)) {
              gain = (t1.distance(t2) + t4.distance(t3) + t3.distance(t5)) -
                  (t1.distance(t3) + t3.distance(t2) + t4.distance(t5));
              if (gain > 0) {
                bestLength -= gain;
                swap();

                improved = true;
                break;
              }
            }
          }
        }
        t1 = suc(t1);
      }
      while (t1 != firstNode);
    }
    while (improved);
  }
}
```

E.13 TwoOpt.java

```
package tsp.optimization;

```

```
import java.util.*;
import Node.*;

/**
 * This class takes a tour and does a 2-opt local search on it.
 * @author Jakob Kierkegaard & Jean-Luc Ngassa
 */
public class TwoOpt {

  public double bestLength;
  private Node t1, t2, t3, t4;
  public LinkedList<Node> nodeTour = new LinkedList<Node>();
  private final LinkedList<Node> temp1 = new LinkedList<Node>();
  private final LinkedList<Node> temp2 = new LinkedList<Node>();

  /**
   * The constructor instantiates the distance of the tour and the tour
   * itself.
   * @param tour The tour that needs optimizing.
   * @param bestLength The distance of the tour that needs optimizing.
   */
  public TwoOpt(LinkedList<Node> tour, double bestLength) {
    this.bestLength = bestLength;
    this.nodeTour = tour;

    twoOpt();
  }

  /**
   * Gets the node before a specific node. If the specific node is in
   * the bottom of the list, it gets the last item in the list, else it
   * gets the node at the index before.
   * @param curNode The node to which the found previous node belongs.
   * @return The previous node.
   */
  private Node pred(Node curNode) {
    if (nodeTour.indexOf(curNode) == 0)
      return nodeTour.getLast();
    else {
      return nodeTour.get(nodeTour.indexOf(curNode) - 1);
    }
  }

  /**
   * Gets the node after a specific node. Gets the node in the index
   * after, and the modulo takes care of the scenario where the
   * specified node is in the end of the list.
   * @param curNode The node to which the found succeeding node belongs.
   * @return The succeeding node.
   */
  private Node suc(Node curNode) {
    return nodeTour.get((nodeTour.indexOf(curNode) + 1) % nodeTour.size());
  }

  /**
   * Reverses the order of nodes in a LinkedList.
   * @param tour The list of nodes that needs to be reversed.
   * @return The reversed list.
   */
  private LinkedList<Node> reverseTour(LinkedList<Node> tour) {
    LinkedList<Node> temp = new LinkedList<Node>();
    for (int i = tour.size() - 1; i >= 0; i--) {
      temp.add(tour.get(i));
    }
    return temp;
  }

  /**
   * The method performs a swap on the tour when the algorithm has
   * found a possible swap.
   * It has to take into consideration if T2 is before or after T4
   * and if T1 is before or after T3 so it knows whether to insert
   * them backwards or forward to give the correct result in the end.
   */
```

```
  private void swap() {
    temp1.clear();

    if (nodeTour.indexOf(t4) < nodeTour.indexOf(t2)) {
      int i = nodeTour.indexOf(t2);
      do {
        if (i == nodeTour.size()) {
          i = 0;
        }
        temp1.add(nodeTour.get(i % nodeTour.size()));
        i++;
      }
      while (i != nodeTour.indexOf(t4));
      temp1.add(t4);
    }
    else {
      for (int i = nodeTour.indexOf(t2); i <= nodeTour.indexOf(t4); i ++) {
        temp1.add(nodeTour.get(i));
      }
    }

    temp2.clear();
    if (nodeTour.indexOf(t3) > nodeTour.indexOf(t1)) {
      int j = nodeTour.indexOf(t3);
      do {
        temp2.add(nodeTour.get(j));
        j = (j + 1) % nodeTour.size();
      }
      while (j != nodeTour.indexOf(t1));
      temp2.add(t1);
    }
    else {
      for (int j = nodeTour.indexOf(t3); j <= nodeTour.indexOf(t1); j ++) {
        temp2.add(nodeTour.get(j));
      }
    }
    nodeTour.clear();
    nodeTour.addAll(reverseTour(temp1));
    nodeTour.addAll(temp2);
  }

  /**
   * Gives a string representation of the tour.
   * @param tour The tour as a LinkedList.
   * @return The string representation of the tour.
   */
  public String printTour(LinkedList<Node> tour) {
    String output = "[";
    for (Node n : tour) {
      output += " " + n.nodeID + ",";
    }
    output += "]";
    return output;
  }

  /**
   * This method goes through the list of nodes to look for any feasible
   * swaps. It gets the T1 and T2 nodes, and goes through all the neighbors
   * of T2 to find a possible swap. If the gain is positive, a swap
   * is done.
   * As the tour changes radically after each swap, the entire tour is
   * checked from the start again after each swap.
   */
  private void twoOpt() {
    boolean improved;
    double gain;
    Node firstNode = t1 = nodeTour.get(0);
    do
      t1.setNeighbors(nodeTour);
    while ((t1 = suc(t1)) != firstNode);
    do {
      improved = false;
      t1 = firstNode;

```

```
        do {
          t2 = suc(t1);
          int dist12 = t1.distance(t2);
          for (Neighbor n : t2.neighbors) {
            t3 = n.toNode;
            t4 = pred(t3);
            gain = (dist12 + t3.distance(t4)) - (t1.distance(t4) + t2.distance(t3));
            if (gain > 0) {
              bestLength -= gain;
              swap();
              improved = true;
              break;
            }
          }
          t1 = suc(t1);
        }
        while (t1 != firstNode);
      }
      while (improved);
    }
}
```