



Jaypee University of Information Technology
Solan (H.P.)
LEARNING RESOURCE CENTER

Acc. Num. *SP07012* Call Num:

General Guidelines:

- ◆ Library books should be used with great care.
- ◆ Tearing, folding, cutting of library books or making any marks on them is not permitted and shall lead to disciplinary action.
- ◆ Any defect noticed at the time of borrowing books must be brought to the library staff immediately. Otherwise the borrower may be required to replace the book by a new copy.
- ◆ The loss of LRC book(s) must be immediately brought to the notice of the Librarian in writing.

Learning Resource Centre-JUIT



SP07012

Real Time Implementation of Ant Colony Optimization

Abhishek Dullu (071266)

Atul Prakash (071269)

Gagandeep Chawla (071334)

Richa Agarwal (071339)

Under the Supervision of

Dr. Satish Chandra



Submitted in partial fulfillment of the Degree of

Bachelor of Technology

Department of Computer Science Engineering

JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY

WAKNAGHAT

SOLAN, HIMACHAL PRADESH

2011



TABLE OF CONTENTS

Chapter No.	Topics	Page No.
	Certificate from the Supervisor	2
	Acknowledgement	3
	Summary	4
Chapter 1	Introduction	5
Chapter 2	An Ant Colony Framework	8
Chapter 3	The TSP	12
Chapter 4	Local search optimization	17
Chapter 5	Implementation	23
Chapter 6	Results and Conclusion	36
Appendices		39
References		59

CERTIFICATE

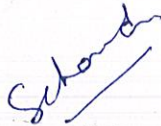
This is to certify that the work titled “**Real Time Implementation of Ant Colony Optimization**” submitted by “**Abhishek Dullu, Atul Prakash, Gagandeep Chawla, Richa Agarwal**” in partial fulfillment for the award of degree of **B. Tech of Jaypee University of Information Technology, Wagnaghat** has been carried out under my supervision. This work has not been submitted partially or wholly to any other University or Institute for the award of this or any other degree or diploma.

Signature of Supervisor

Name of Supervisor

Designation

Date



Dr. Satish Chandra

Associate Professor

ACKNOWLEDGEMENT

We first and foremost want to thank **Retd. Brig. S.P Ghrera** without whose help and support this project would not have been possible at all.

I would like to acknowledge **Dr. Satish Chandra** for his gracious support and guidance in this project. We appreciate his immense help and his feedbacks which helped us in developing the project so successfully.

Abhishek Dullu

Atul Prakash

Gagandeep Chawla

Richa Agarwal

SUMMARY

The project necessitated a thorough study and implementation of Ant Colony Optimization technique. As such the main focus of the project has been to solve a combinatorial problem.

We have chosen to solve Travelling Salesman Problem (TSP) using Ant Colony Optimization technique. In this problem we have a given set of cities. A salesman has to visit each city once in such a way that the tour takes minimum time.

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 achieve better tour constructions.

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.

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.

1.1 Motivation

These new techniques can sometimes be used in modern day technology to solve problems that might be too time-consuming. 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

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.

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 Experiment¹ 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. τ_{AB} and τ_{AC} are defined following a random distribution of a constant δ over $[0,1]$.

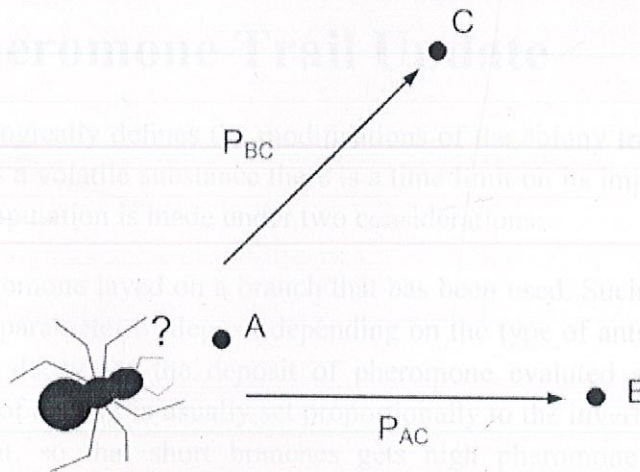


Figure 2.1: An ant's choice.

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. $\forall i$ th as ant id, we define the following expression:

$$\tau_{AB_{i+1}} = \begin{cases} \tau_{AB_i} + 1, & \delta \leq P_{AB} \\ \tau_{AB_i}, & \delta > P_{AB} \end{cases} \quad (2.1)$$

The probability $\tau_{AC_{i+1}} = \begin{cases} \tau_{AC_i} + 1, & \delta \leq P_{AC} \\ \tau_{AC_i}, & \delta > P_{AC} \end{cases}$ (2.2) the next state position in the environment. It is enunciated by:

$$P_{AB} = \frac{\tau_{AB}}{\tau_{AB} + \tau_{AC}} \quad (2.3)$$

(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 there inforcement 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 choosing their path following a random probability. Constraint settings make an ant's choice of moves to be more constructive rather than a total dependence 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].

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:

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

3.1 A heuristic tour construction

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, where nodes are not visited twice.

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.

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 was no consideration of the return of the ant which led to unrealistic results.

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.

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

$$j = \begin{cases} \operatorname{argmax}_{u \in J_i^k} \{[\tau_{iu}(t)] \cdot [\eta_{iu}]^\beta\} & \text{if } q \leq Q_0 \\ J & \text{if } q > Q_0 \end{cases}$$

$\eta_{i,u}$ 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^k

i , the set of cities to visit by ant k at city i or the choice list.

$u \in J^k$

i , 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.

$$P_{i,J}^k(t) = \frac{[\tau_{i,J}(t)]^\alpha \cdot [\eta_{i,J}]^\beta}{\sum_{u \in J_i^k} [\tau_{i,u}(t)]^\alpha \cdot [\eta_{i,u}]^\beta}$$

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:

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

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

$$\tau_0 = (n \cdot L_{nn})^{-1}$$

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.

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

ξ 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:

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

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

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

Where L_{best} is the length of the best cycle, and W is a parameter ranged between [1 100]. Our choice of 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 2, meaning that distances (or weights) between nodes are expressed on an Euclidean EUC-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 four optimizations and implemented methods for local search based on the 2-opt and 2,5-opt algorithms.

Therefore it will always be an advantage to set an optimization method for improving an initial result from the TSP algorithm for implementation. The most common way of optimizing the TSP is local search.

The most commonly used optimization algorithms for the TSP are 2-opt, 2,5-opt, and Lin-Kernighan. The Lin-Kernighan algorithm is generally seen as being the most efficient optimization algorithm right now, particularly after gold fields and cross-country optimization was implemented in 1991. It is very hard to beat in its complexity, which was not 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 resources to implement it, so we turned ourselves towards implementing 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 that 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 part of the local search optimization techniques, 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 better optimized (if not the optimized) tour, the 2-opt is a feasible choice when looking at its results versus the time required to obtain those results.

The implementation of the 2-opt is based on the following points, as shown in figure 4.1:

1. Take two nearest consecutive nodes, pairs A & B and C & D from a tour.
2. Check to see if the sum of edges A-B and C-D is greater than the sum of edges A-C and B-D.
3. If that is the case, remove A and B, reinsert them reversely between C and D.

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 distinct ways:

Local search optimization

No matter how efficient 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 implementing 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 one 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 lose a potential good improvement.

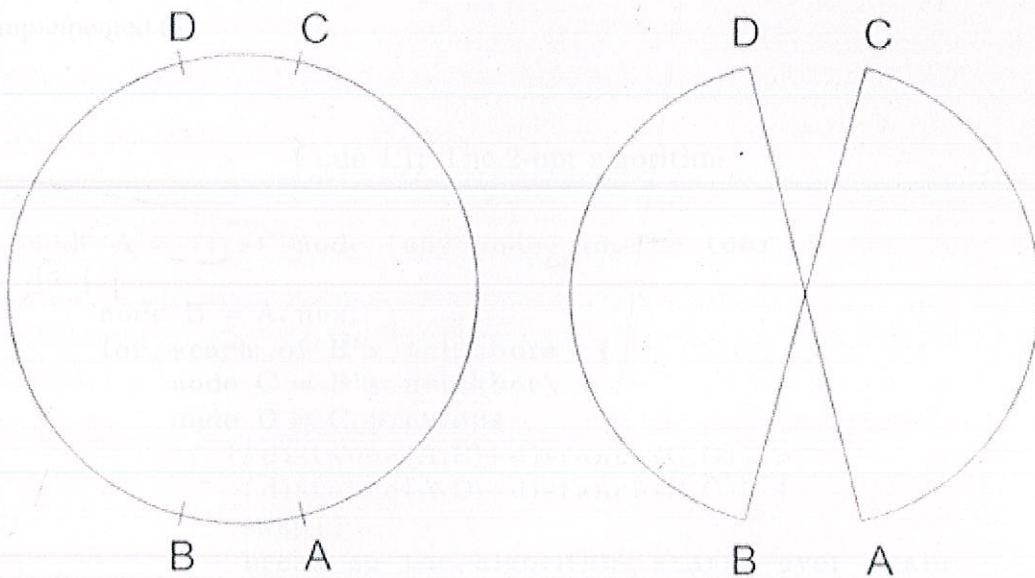


Figure 4.1: A 2-opt operation.

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 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

```
1 do {
2   node A = first node (any node) in the tour
3   do {
4     node B = A.next
5     for (each of B's neighbors) {
6       node C = B's neighbor
7       node D = C.previous
8       if ((distance(A,B)+distance(C,D)) >
9           (distance(A,D)+distance(B,C)) {
10        swap()
11        break so the algorithm starts over again
12      }
13    }
14    A = A.next
15  } while (A != first node)
16 } while (there has been made changes)
```

Code 4.2: The 2-opt's swap algorithm

```
1 node temp
2 for (all the nodes from A to C){
3     temp = node.next
4     node.next = node.previous
5     node.previous = temp
6 }
```

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 improvement 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 algorithm 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 $CE + AD + BD$.

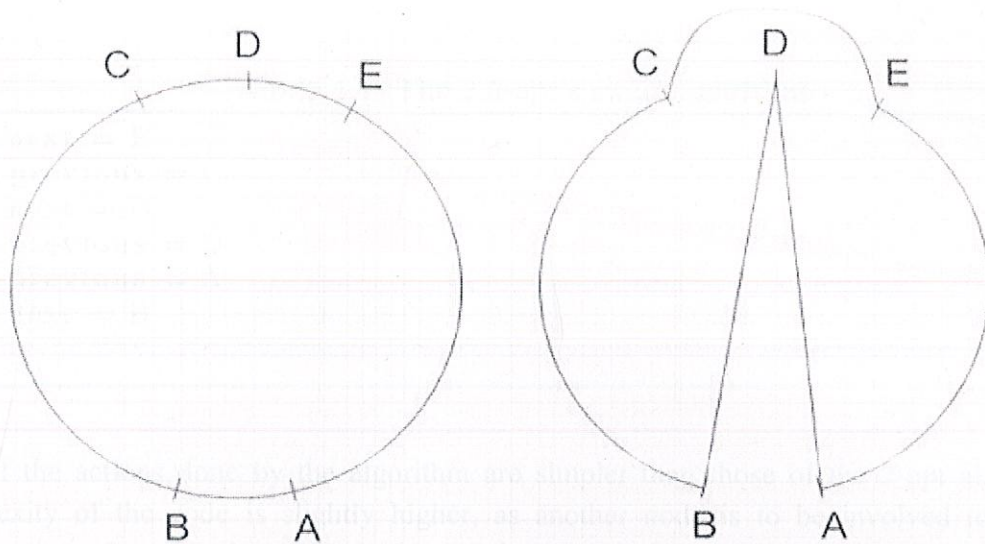


Figure 4.2: A 2,5-opt operation.

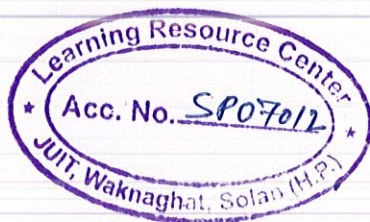
Its simplicity can be seen in the code fragment 4.3.

Code 4.3: The 2,5-opt algorithm

```

1 do {
2   node A = first node (any node) in the tour
3   do {
4     node B = A.next
5     for (each of A's neighbors) {
6       node D = B's neighbor
7       node C = D.previous
8       node E = D.next
9       if ((distance(A,B)+distance(C,D)+distance(D,E)) >
10          (distance(A,D)+distance(B,D)+distance(C,E)) {
11         swap()
12         break so the algorithm starts over again
13       }
14     }
15     A = A.next
16   } while (A != first node)
17 } while (there has been made changes)

```



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

```
1 C.next = E
2 E.previous = C
3 A.next = D
4 B.previous = D
5 D.previous = A
6 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.

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 SP6 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 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,

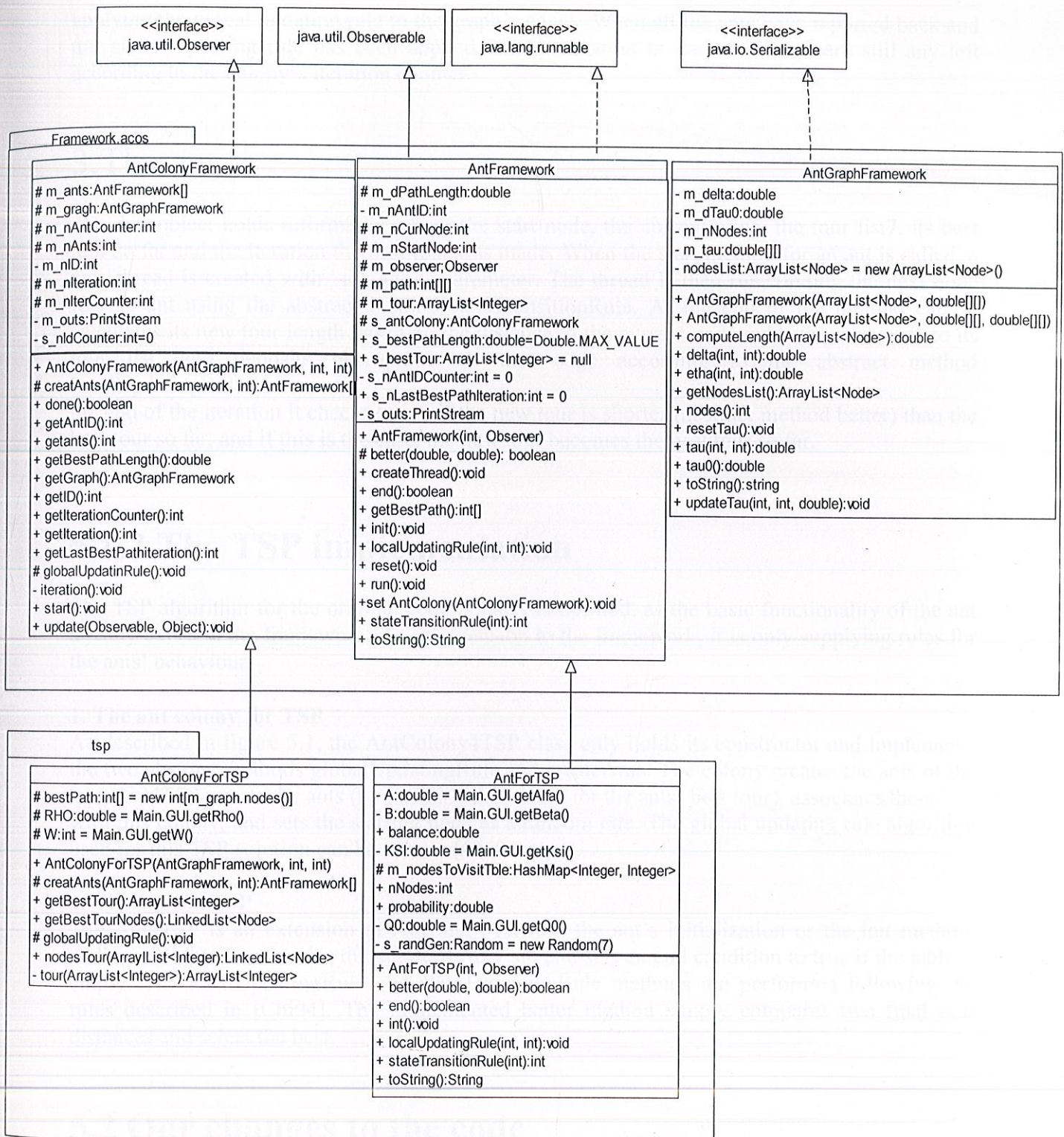


Figure 5.1: UML diagram of the old version of the program.

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, 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 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.

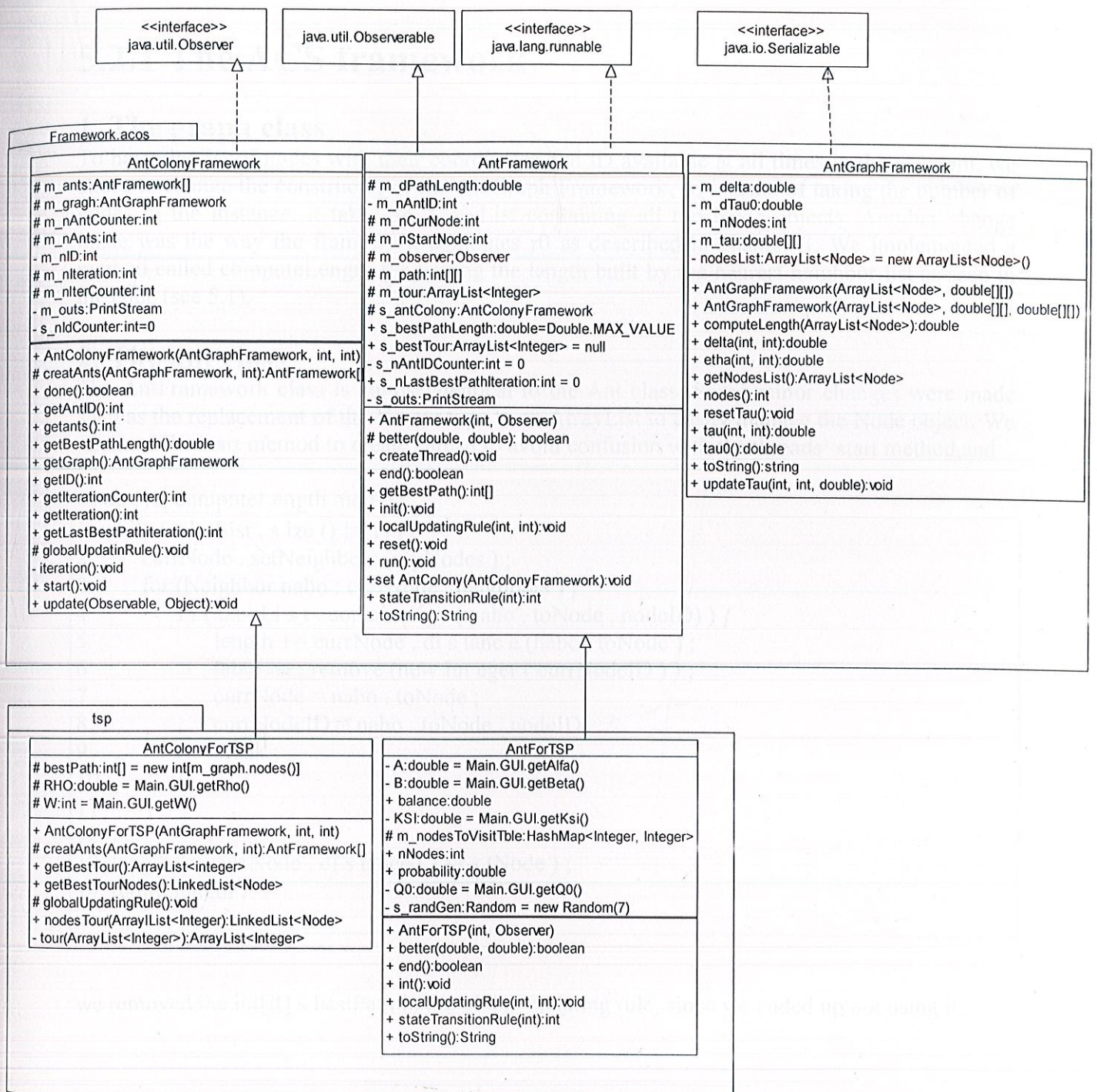


Figure 5.2: UML diagram of the new version; the framework. Aco's and tsp packages.

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 AntGraph Framework, 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 method to createThread to avoid confusion with the threads' start method, and

Code 5.1: computeLength method

```
1 while ( tabuList . s i z e () != 1 ) {
2     currNode . s e t N e i g h b o r s ( m y N o d e s ) ;
3     f o r ( N e i g h b o r n a b o : c u r r N o d e . n e i g h b o r s ) {
4         i f ( t a b u L i s t . c o n t a i n s K e y ( n a b o . t o N o d e . n o d e I D ) ) {
5             l e n g t h + = c u r r N o d e . d i s t a n c e ( n a b o . t o N o d e ) ;
6             t a b u L i s t . r e m o v e ( n e w I n t e g e r ( c u r r N o d e I D ) ) ;
7             c u r r N o d e = n a b o . t o N o d e ;
8             c u r r N o d e I D = n a b o . t o N o d e . n o d e I D ;
9             b r e a k ;
10        }
11    }
12 }
13 l e n g t h + = c u r r N o d e . d i s t a n c e ( f i r s t N o d e ) ;
14 r e t u r n l e n g t h ;
```

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 access 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 accommodate the Node objects instead of Integers, 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 ζ 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 neighbor 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 Integers being node IDs, or actual Node objects. The global updating rule method has been changed

Code 5.2: The local updating rule

```
1 public void localUpdatingRule ( int nCurNode , int nNextNode) {
2     final AntGraphFramework graph = s antColony . getGraph ( ) ;
3     double val = ( ( double ) 1 - KSI) * graph . tau ( nCurNode ,
4                 nNextNode) + ( KSI * ( graph . tau0 ( ) ) ) ;
5     graph . updateTau(nCurNode , nNextNode , val ) ;
6 }
```

Code 5.3: The global updating rule

```
1 protected void globalUpdatingRule ( ) {
2     double dEvaporation ;
3     double dDeposit ion ;
4     for ( int i = 0 ; i < m graph . nodes ( ) ; i ++ )
5         bestPath [ i ] = AntForTSP . getBestPath ( ) [ i ] ;
6     for ( int r = 0 ; r < m graph . nodes ( ) ; r ++ ) {
7         for ( int s = r + 1 ; s < m graph . nodes ( ) ; s ++ ) {
8             for ( int i = 0 ; i < super.getAnts ( ) ; i ++ ) {
9                 double deltaTau =
10                    ( W / AntForTSP.s_dBestPathLength ) ;
11                 dEvaporation = ( ( double ) 1 - RHO) *
12                    m graph . tau ( bestPath [ r ] , bestPath [ s ] ) ;
13                 dDeposit ion = RHO * deltaTau ;
14                 m graph . updateTau( bestPath [ r ] , bestPath [ s ] ,
15                    dEvaporation + dDeposit ion ) ;
```



```
16         }
17     }
18 }
19 }
```

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.

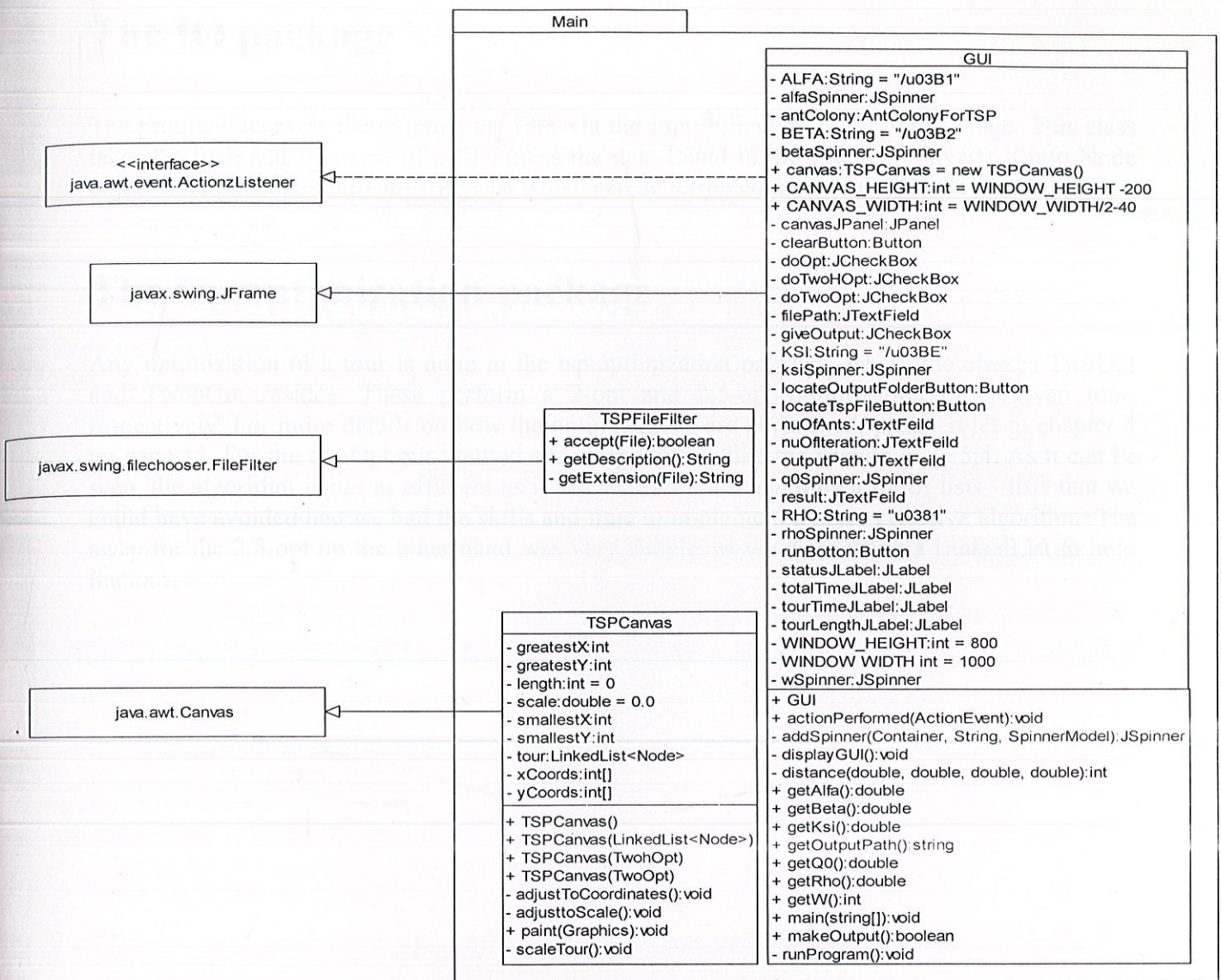


Figure 5.3: UML diagram of the new version; the Main package

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:

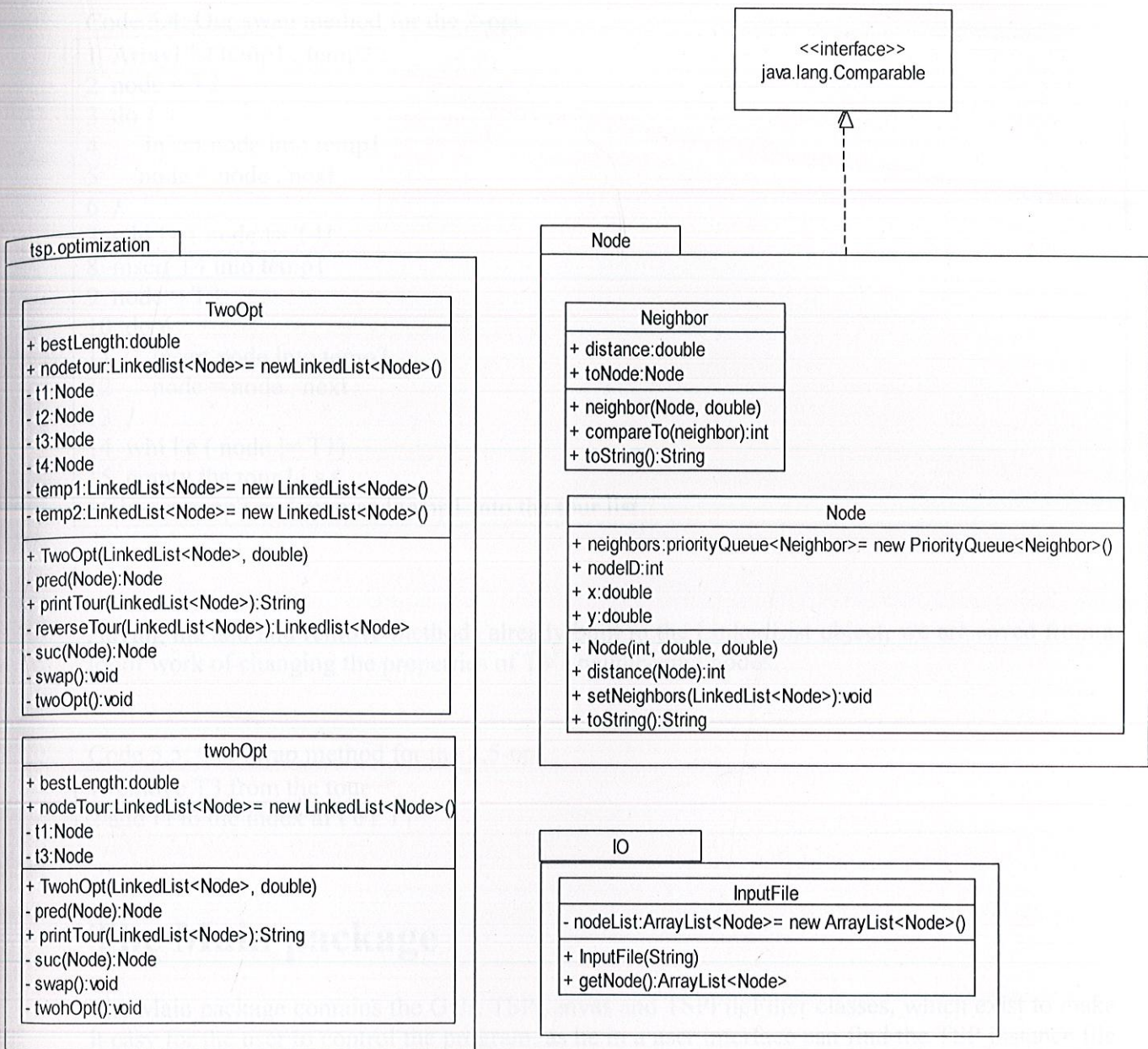


Figure 5.4: UML diagram of the new version; the Node, IO and tsp.optimization packages.

Code 5.4: Our swap method for the 2-opt

```
1 ArrayList temp1 , temp2 ;
2 node = T2
3 do {
4     insert node into temp1
5     node = node . next
6 }
7 while ( node != T4)
8 insert T4 into temp1
9 node = T3
10 do {
11     insert node into temp2
12     node = node . next
13 }
14 while ( node != T1)
15 empty the tour list
16 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.

```
1 remove T3 from the tour
2 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 , Q_0 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 class 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, without getting stretched.

Results and Conclusion

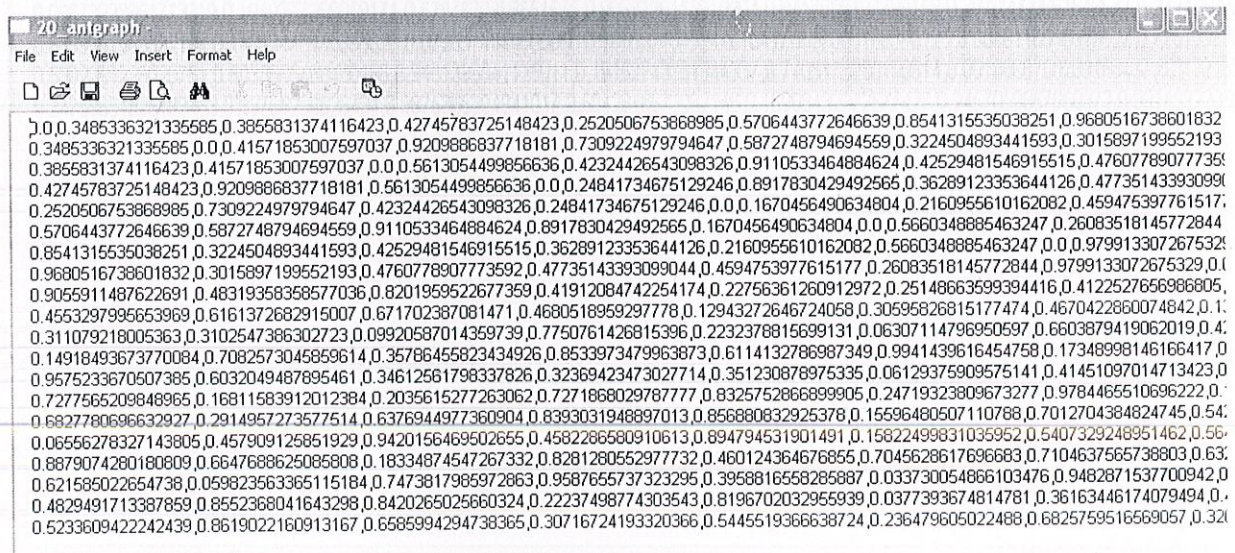
We took Chirico's original program and changed it so it supports node objects instead of integers, and we applied a few changes in the structure based on Dorigo's implementation of the TSP. Unfortunately we did not have time for testing different combinations of the program's parameters, trying to find a golden combination that would give us a better result than other combinations, and instead we decided to go with values recommended by Dorigo.

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.

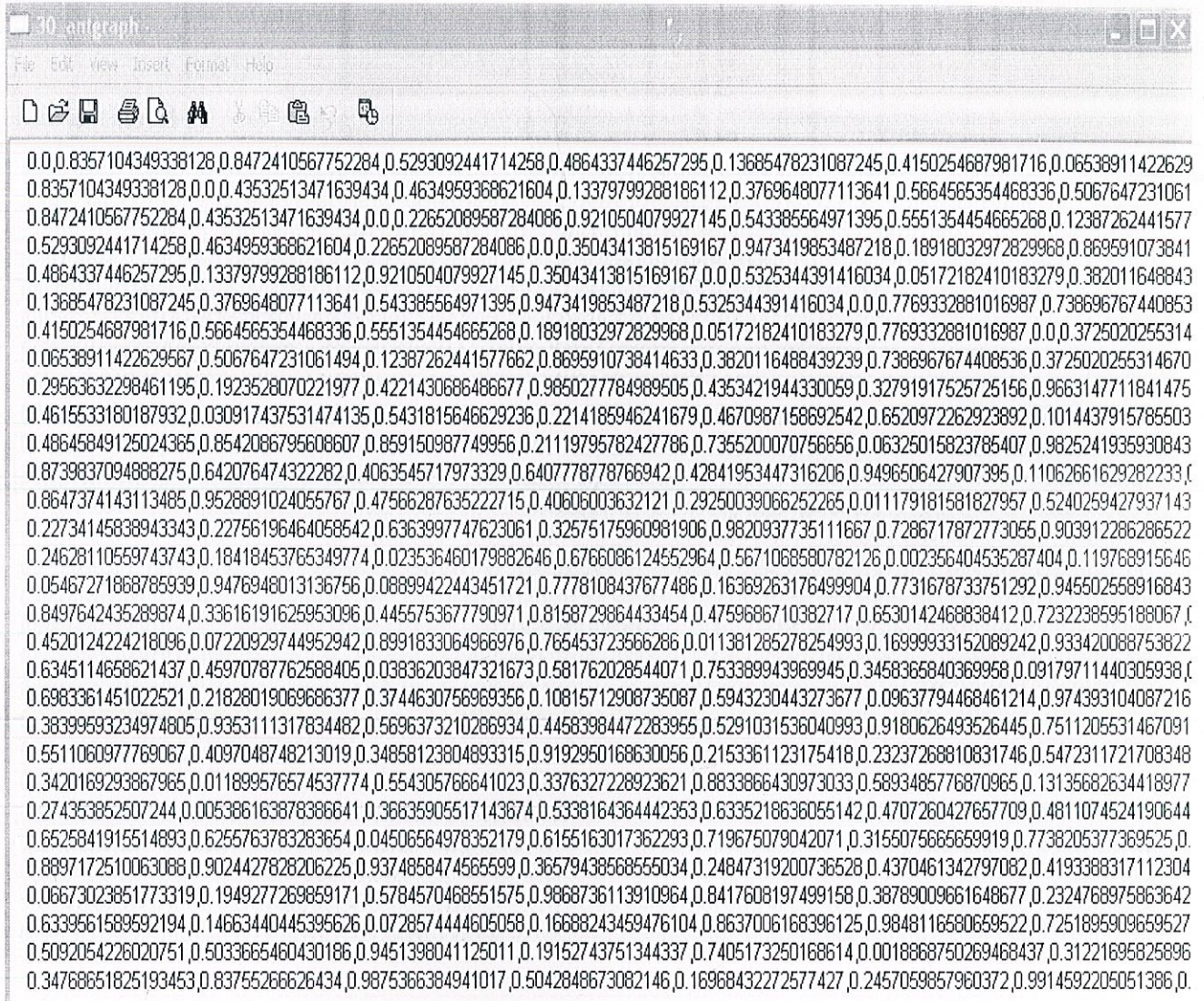
Output using different set of inputs.

No. of cities	No. of Ants	Iteration	Best tour length
20	10	50	2.5118568295153025
30	10	50	1.5501922784261288
50	10	50	2.130735899811355

Randomly generated graph for input set 1.



Randomly generated graph for input set 2.



Randomly generated graph for input set 3.

```
50_anigraph
File Edit View Insert Format Help
[Icons]
0.0,0.07890233783448031,0.7896312947525304,0.7911140529771883,0.043708966972266894,0.04788687661062474,0.0799354140146673,0.5759727806047165,0.8180260431
0.07890233783448031,0.0,0.1793364063761651,0.18554710451456302,0.4537467497240887,0.7145781321265343,0.5630258975887685,0.6543676646816613,0.773902452271
0.7896312947525304,0.1793364063761651,0.0,0.34445694052717335,0.7333851288023813,0.852268219977618,0.48621591536719044,0.7166202463388934,0.6630916243271
0.7911140529771883,0.18554710451456302,0.34445694052717335,0.0,0.18434093615724856,0.09241150911530893,0.5373814881452335,0.9403628331316188,0.4106491301
0.043708966972266894,0.4537467497240887,0.7333851288023813,0.18434093615724856,0.0,0.35306292927052996,0.526599935499803,0.859218144326366,0.58570538054
0.04788687661062474,0.7145781321265343,0.852268219977618,0.09241150911530893,0.35306292927052996,0.0,0.927578965910961,0.2866401693101309,0.4054239883961
0.0799354140146673,0.5630258975887685,0.48621591536719044,0.5373814881452335,0.526599935499803,0.927578965910961,0.0,0.1323100650327078,0.01620818479856
0.5759727806047165,0.6543676646816613,0.7166202463388934,0.9403628331316188,0.859218144326366,0.2866401693101309,0.1323100650327078,0.0,0.88672800686837
0.81802604305047,0.7739024522706406,0.6630916243276372,0.4106491300109735,0.5857053805445238,0.4054239883960984,0.016208184798564784,0.8867280068683735
0.4736372771640839,0.6870592539227149,0.17319274558354214,0.7338401376297463,0.5059853859426299,0.004596068571310341,0.4701572321576738,0.5123997493610
0.8997960910465322,0.6274355270330517,0.8699365497534429,0.22846568622135754,0.09835282377851295,0.5181688689732488,0.7953676954144844,0.31651648881197
0.9701647390487089,0.7740956802657886,0.3779815867250852,0.6496855009349052,0.1912340082340438,0.4525371694784155,0.15758992743549427,0.5333529995160327
0.51904734999307928,0.9618356212937024,0.9967122083398205,0.33833652069477427,0.2348313549660449,0.14902870338203955,0.48908942313645654,0.2982061091945
0.619396863307225,0.7686461902302424,0.9890966522237941,0.7717457345967096,0.4961067808608225,0.27054755882466763,0.22277593682422003,0.629528652953518
0.41812768855588024,0.4980131127244465,0.753570667284431,0.35006541209231434,0.2608249994342101,0.739278413455799,0.8573449245500617,0.1752213339366101
0.8940295625807041,0.9159579239648256,0.7320006898454569,0.3301122349441984,0.9508959077143143,0.22464541160726137,0.6336016823198241,0.303121624064945
0.8748541626873347,0.7114178171447304,0.29620849298547425,0.00476769552147464,0.8764874123142459,0.03382320450404952,0.02103355540991292,0.81924953856
0.8187998901009814,0.038179378099927885,0.6536558590051395,0.5879235090300177,0.4951668138122882,0.3440320324301257,0.5333862306114011,0.45109049058092
0.7999286776325013,0.290930996334822,0.5086679341265526,0.017222295139413424,0.1940104840252348,0.4105324087175716,0.09080740837004997,0.11485651542723
0.2834598641365463,0.4329502379226555,0.9415040184356825,0.645131129882785,0.24306184552209764,0.42690580565015656,0.4319141175814213,0.082266979909635
0.435444771510782,0.7539927081763101,0.7870574481342753,0.3447880925885348,0.4890355050358941,0.903184388716907,0.43946929683529745,0.8730664713209768
0.9841932091468936,0.8574783229655264,0.6620775229666366,0.5048516881762017,0.7576924285717588,0.269741247075326,0.9882189273702366,0.19359709893612154
0.32103956551838897,0.3532364583220049,0.35725902132417486,0.46631697098847247,0.6412732785640908,0.23067266893020044,0.20904425169267715,0.23012659568
0.8566371840120027,0.5986882325199281,0.0427634806524525,0.7497463716980506,0.7861054762119211,0.29661301752108393,0.16655787364212338,0.13725826879802
0.36024689801533727,0.3332320370590869,0.31540893510332246,0.10375832212563252,0.4741827139869811,0.4072425368497471,0.09017658395594819,0.248661296734
0.26481925354527946,0.30021168779033214,0.9133972049989374,0.4597333305625274,0.20399732257264724,0.8537152518474986,0.6088341968742045,0.2933163229364
```


Appendices A

Screenshots



File input

The screenshot shows a software application window titled "TSP output". The interface is divided into several sections:

- Parameters:** A table with two columns. The first column lists parameters: α , β , W , Q_0 , P , ξ . The second column contains values: 0.1, 2, 1, 0.8, 0.1, 0.1.
- File settings:** A section with a "Locate TSP file" label and a checkbox labeled "Do you want intermediate results printed to files?".
- Optimization:** A section with a "RUN" button, a "Clear settings" button, and a "Use default values" button.
- Status:** A section at the bottom of the main window.

An "Open" file dialog is overlaid on the main window. It shows the following details:

- Look in:** Documents
- File list:** Bluetooth Exchange Folder, Dell WebCam Central, Downloads, GOM Player, My Library, Vuze Downloads.
- File name:** (empty text field)
- Files of Type:** TSP files
- Buttons:** Open, Cancel

The Windows taskbar at the bottom shows the Start button, several application icons, and the system tray with the date and time set to 01:05.

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 Euclidian 2D which we have used for benchmarking.

C Sample Codes

C.1 AntColonyForTSP.java

```
import java.util.*;
import Node.Node;
public class AntColonyForTSP extends AntColonyFramework
{
protected static final double RHO=Main.GUI.getRho();
protected static final int W= Main.GUI.getW();
int[] bestPath= new int[m_graph.nodes()];
public AntColonyForTSP(AntGraphFramework graph, int ants, int iterations)
{
super(graph, ants, iterations);
}
public 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;
}
public 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;
m_graph.updateTau(bestPath[r], bestPath[s], dEvaporation + dDeposition);
m_graph.updateTau(bestPath[s], bestPath[r], dEvaporation + dDeposition);
}
}
}
}
```

```

public LinkedList<Node> getBestTourNodes()
{
return nodesTour(AntFramework.s_bestTour);
}
public ArrayList<Integer> getBestTour()
{
return tour(AntFramework.s_bestTour);
}
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;
}
public static ArrayList<Integer> tour(ArrayList<Integer>tour)
{
for(int i=0;i<tour.size();i++)
{
tour.set(i,tour.get(i)+1);
}
return tour;
}
}

```

C.2 AntColonyFramework.java

```
import java.util.*;
import java.io.*;
public abstract class AntColonyFramework implements Observer
{
    private PrintStream m_outs;
    protected final AntGraphFramework m_graph;
    protected AntFramework[ ] m_ants;
    protected final int m_nAnts;
    protected int m_nAntCounter;
    protected int m_nIterCounter;
    protected final int m_nIterations;
    private final int m_nID;
    private static int s_nIDCounter=0;
    public AntColonyFramework(AntGraphFramework graph,int nAnts, int Iterations)
    {
        m_graph = graph;
        m_nAnts = nAnts;
        m_nIterations = nIterations;
        s_nIDCounter++;
        m_nID = s_nIDCounter ;
    }
    public synchronized void start()
    {
        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();
            }
        }
        while(m_nIterCounter<m_nIterations)
```

```

{
if(m_nIterCounter%5==0)
System.out.println("iteration:" +(m_nIterCounter +1));
iteration();
try
{
wait();
}
catch(InterruptedException ex)
{
ex.printStackTrace();
}
synchronized(m_graph)
{
globalUpdatingRule();
}
}
if(Main.GUI.makeOutput())
{
if(m_nIterCounter==m_nIterations)
{
m_outs.close();
}
}
}
public void iteration()
{
m_nAntCounter=0;
m_nIterCounter++;
for(AntFramework ant:m_ants)
ant.createThread();
}
public AntGraphFramework 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 int GetAntID()
{
return m_nAntCounter;
}
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();
}
}
public double getBestPathLength()
{
return AntFramework.s_dBestPathLength;
}
public int getLastBestPathIteration()
{
return AntFramework.s_nLastBestPathIteration;
}
public final boolean done()
{
return m_nIterCounter==m_nIterations;
}
protected abstract AntFramework[] createAnts(AntGraphFramework graph, int ants);
protected abstract void globalUpdatingRule();}

```


C.3 GUI.java

```
import java.util.*;
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;
public class GUI extends JFrame implements ActionListener
{
    private static AntColonyForTSP antColony;
    public static TSPCanvas canvas=new TSPCanvas();
    public static JTextField filePath, outputPath, nuOfAnts, nuOfIterations, result;
    private Button rnButton, 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;
    public GUI()
    {
        displayGUI ();
    }
    public 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));
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");
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");
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);
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"));
class intVerifier extends InputVerifier
{
public final boolean verify (JComponent comp)
{
boolean returnValue= false;
JTextField textField = (JTextField)comp;
try
{
if(Integer.parseInt(textField.getText())>0)
return Value = true;
else
Toolkit.getDefaultToolkit().beep();
}
catch (NumberFormatException e)
{

```

```

Toolkit.getDefaultToolkit().beep();
}
return returnValue ;
}
}

```

```

nuOfAnts.setInputVerifier(new intVerifier());
nuOfIterations.setInputVerifier (new intVerifier());
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);
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);
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);
extraJPanel= new JPanel(new GridLayout(2,1));
extraJPanel.add(optimizationJPanel);
extraJPanel.add(statusJPanel);
innerPJPannel.add(numberJPanel);
innerPJPannel.add(fileJPanel);
innerPJPannel.add(extraJPanel);
parameterJPanel.add(buttonJPanel,"South");
parameterJPanel.add(innerPJPannel);
this.add(parameterJPanel);
this.add(visualJPanel);
this.setVisible(true);
canvas.repaint();
}
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);
}
}
else
if(e.getSource()==runButton)
{
missingParameters="";
if(nuOfAnts.getText().equals("")||nuOfIterations.getText().equals("")||filePath.getText().equals("")||
giveOutput.isSelected()&&outputPath.getText().equals(""))|(doOpt.isSelected()&&!doTwoOpt.isS
elected()&&!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
{
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\nDo 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.getAbsolutePath().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.getAbsolutePath().toString());
}
}

```



```

}
}
public 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;
}
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);
}
}
}
}

```

```

public 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);
}
public static void main(String [ ] args)
{
new GUI();
}
public static double getBeta()
{
return Double.parseDouble(betaSpinner.getValue().toString());
}
public static double getRho()
{
return Double.parseDouble(rhoSpinner.getValue().toString());
}
public static double getQ0()
{
return Double.parseDouble(q0Spinner.getValue().toString());
}
public static int getW()
{
return Integer.parseInt(wSpinner.getValue().toString());
}
public static double getAlfa()
{
return Double.parseDouble(alfaSpinner.getValue().toString());
}
public static double getKsi()
{
return Double.parseDouble(ksiSpinner.getValue().toString());
}
public static boolean makeOutput()
{
return giveOutput.isSelected();
}
public static String getOutputPath()
{
return outputPath.getText()+"\\";
}}

```

References

- Marco Dorigo , Mauro Birattari, and Thomas Stützle, Ant Colony Optimization, Ants as a Computational Intelligence Technique, Université Libre de Bruxelles, BELGIUM.
- Chirico, Chirico's framework for Ant Colony Optimization.
- Eric Bonabeau, Marco Dorigo, Guy Theraulaz, Swarm Intelligence From Natural To artificial Systems, A Volume In The Santa Fe Institute Studies, In The Sciences Of Complexity, New York Oxford, 1999.
- www.aco.org
- Christian Blum, Daniel Merkle(Eds.), Swarm Intelligence, Introduction And Applications, Barcelona, Odense , April 2008.
- E . Murat ESIN, Senol Zafer ERDOGAN, Self Cloning Ant Colony Approach and Optimal Path Finding, Turkey.