

Swarm Intelligence

Object Oriented Programming Using C#

Matteo De Cicco

ABSTRACT

Implementation of ACO algorithms with application to the famous Traveling Salesman Problem (TPS).

Table of Contents

Introduction	3
Problem discussion	3
Interpretation of the specification	3
Introduction to UML	5
Use-case Model	5
Use-case View	5
Scenario Description	6
CRC Cards	6
Statechart Diagram	8
Interaction Diagram	9
Analysis Model	10
Class diagram	10
Statechart Diagram	12
Interaction Diagram	12
Design Model	14
Class diagram	14
Details of Algorithm implementation	16
Greedy Algorithm	16
ACO v0 algorithm	17
ACO v1 algorithm	18
ACO v2 algorithm	18
Account of Structured Testing	20
TSPData Class Testing	20
TSPSolver Class Testing	21
Conclusions	24

Introduction

Problem discussion

The Travelling Salesman Problem is one of the most studied problems with NP-hard complexity that are still a great field of research in the Computational Theory. Starting from the second half of the XX^o century, these kind of problems have been approached with the aid of statistic methods based on natural behaviours. As observed in many situations, animals with “limited capacities” and no coordinator such as ants, bees and birds, can solve natural problems, such as finding the nourishment and communicate its position.

Nowadays, in Telecommunication Networks, Data Centers and Big Data Analytics, NP-hard complexity problems are still at the heart of the optimization problems, where a lot of research behind this trying to create algorithms that are capable of minimize the efforts and resources employed.

The aim of this project is to understand the basics on the Swarm Intelligence algorithm and implement a series of algorithms of the Ant Colony Optimization class in order to find a solution for the Travelling Salesman Problem, giving back a tour length that covers of all the cities. Each algorithm implemented will provide a better “intelligence”, based on specific formulas applications and optimization procedures so that it returns a tour length that is smaller than the previous ones.

Interpretation of the specification

The data in a TSP can be interpreted as a graph $G(N,E)$ where the N , “the nodes”, represent all the cities to visit and E , “the arcs”, represents all the distances between those cities. In this case, we will consider the problem with “symmetrical” data, so the distance d from city i to city j will be equal to the one from city j to city i : $d_{ij} = d_{ji}$.

The algorithms, that we will apply, derive from the general class of Swarm Intelligence algorithms introduced before.

In this case, we will base our algorithms on the Ant Colony Optimization class of SI algorithms. This class of algorithms bases its operations on the behaviour of real “ants” inside a swarm, thus modelling the capacity of communicate between ants via a “pheromone deposit” event and a “pheromone evaporation” event. Each colony is made up of a specific number of ants. Those ants will explore the dataset one by one.

The “pheromone deposit” event is the only way the colony has to interact with all its ants; this event will takes place only when the first set of ants (the first colony) completes the exploration, so that the next swarm will find additional information on the dataset and each ant will take decisions on its path based on that.

The ants implemented are “artificial”, so it means that they have specific characteristic not related to the real world ants. In particular they have memory of the cities visited and base their decisions on probability.

The formula for local probability used is:
$$p_{ij}^k = \frac{[\tau_{ij}(t)][\eta_{ij}]^\beta}{\sum_{l \in N_i} [\tau_{il}(t)][\eta_{il}]^\beta} \quad j \in N_i^k$$

The amount of pheromone deposited is:
$$\Delta\tau_{ij}^k(t) = \begin{cases} Q/L^k(t) & \text{if } (i, j) \in T^k(t) \\ 0 & \text{if } (i, j) \notin T^k(t) \end{cases} \quad k = 1..m$$

The process of pheromone deposit and evaporation is given by:
$$\tau_{ij}(t) \leftarrow (1 - \rho)\tau_{ij}(t) + \sum_{k=1}^m \Delta\tau_{ij}^k(t)$$

The parameters in the previous formulas change the weight of each element and need to be chosen during the tuning of the algorithm for improve the “visibility” of the search inside the exploration space.

The first algorithm implemented is the “Greedy” Algorithm.
Using this algorithm the next city visited is always the nearest one and being a deterministic algorithm it doesn’t need to be iterated for more than one colony.

The second algorithm implemented is the basic ACO (ACO v0).
It implements the formulas previously presented. In order to have a correct behaviour the process is iterated for more than one colony, so that the pheromone deposit leads to a better solution than ‘Greedy’.

The third algorithm implemented is the ACO v1, where the pheromone deposit process is improved with a second process that selects the best sequence of cities and adds an extra amount of pheromone.

The fourth algorithm implemented is the ACO v2.
In this algorithm, the visibility of the exploration space is improved with a pseudo-random rule based on the analysis of the local probability.

Introduction to UML

The process of modelling this project using UML is achieved in three consequent phases; each of them is a more accurate analysis of the specifications using different views.

In this case, the package of specification received was really detailed and its context was clearly a 'domain modelling'.

The requirements capture was then based on an analysis of the problem domain.

Functional Requirements identified:

- the System needs to be able to take a human readable file as input
- it must show essential information of the problem
- it must show the results of the computational algorithm

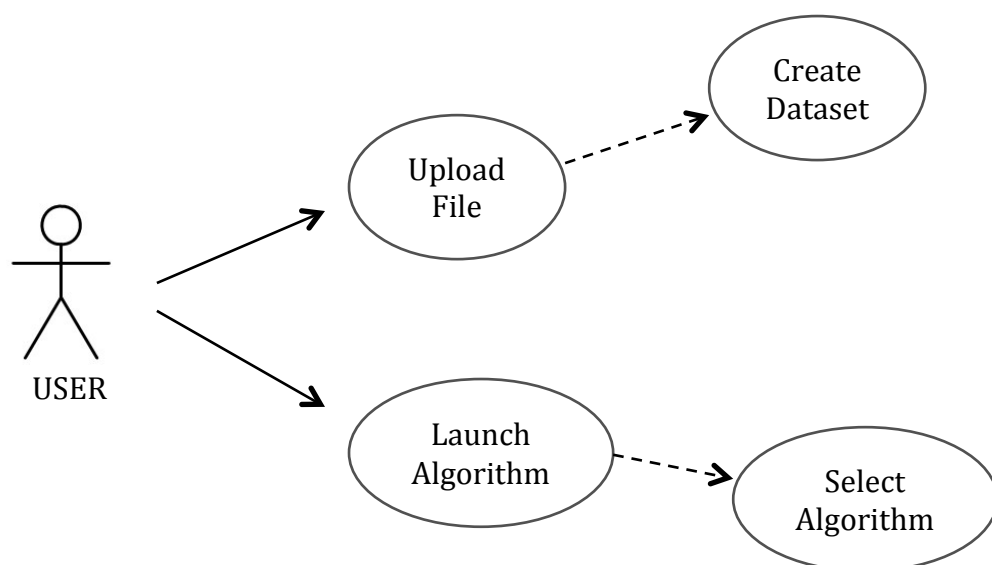
The specifications on the GUI were not considered as part of the software design, however they have been a useful tool for understanding the expected output of the system.

Despite the functional requirements were few and simple, the non-functional ones reflected the very technical nature of this project, such as optimized data structure for fast computation, scalability and speed of response.

After a phase of Requirement Capture, the modelling was carried out through the Use-Case Modelling, the Analysis Modelling and then the Design Modelling.

Use-case Model

Use-case View



Scenario Description

The user selects a file from the computer. The system reads the file and picks information such as the number of the cities, the name of the file, the optimum tour length given, all the cities and their coordinates.

The system then creates a dataset of the arcs that represent the distances from each city to any other.

The user selects the type of algorithm among those available and, at the end, he starts the algorithm computation that will produce as result the length of the tour and its sequence of cities visited in order, using the properties of the “artificial ants” implemented.

CRC Cards

The scenario description helps to understand the design.

Based on this description, it was possible to select the main classes and their role inside the program.

From this analysis, these are the CRC cards produced:

CLASS: TSP Data	
Responsibilities	Collaborators
This class is responsible for reading the file in input, creating a dataset and storing it in a sensible way in order to have it suitable for the solver.	City, Arcs, TSP Solver

CLASS: City	
Responsibilities	Collaborators
This class is responsible for organize the information about a city such as its number, x coordinate and y coordinate.	TSP data, Arc

CLASS: Arc	
Responsibilities	Collaborators
This class is responsible for organize the information about an arc such as the origin city, the destination city and the length of the arc.	TSP data, City

CLASS: TSP Solver	
Responsibilities	Collaborators
This class is responsible for provide the results of the algorithm computation based on the given data and the type of algorithm requested.	TSP data, Algorithm

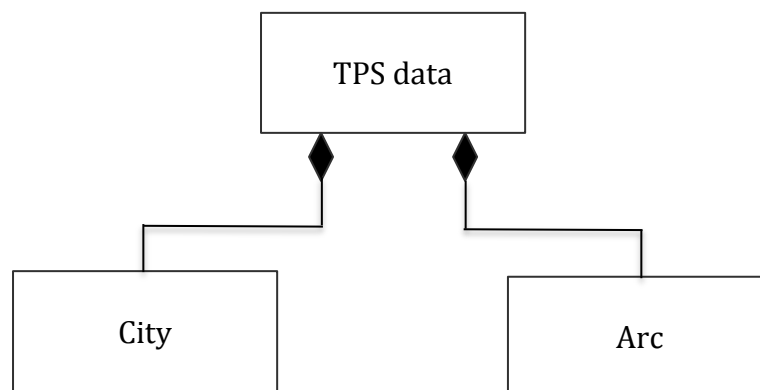
CLASS: Algorithm	
Responsibilities	Collaborators
This class is a general class for the Algorithm implementation. It is responsible for the behaviour of the algorithm and its parameters. Each different type of Algorithm will derived from this class.	TSP Solver, Ant

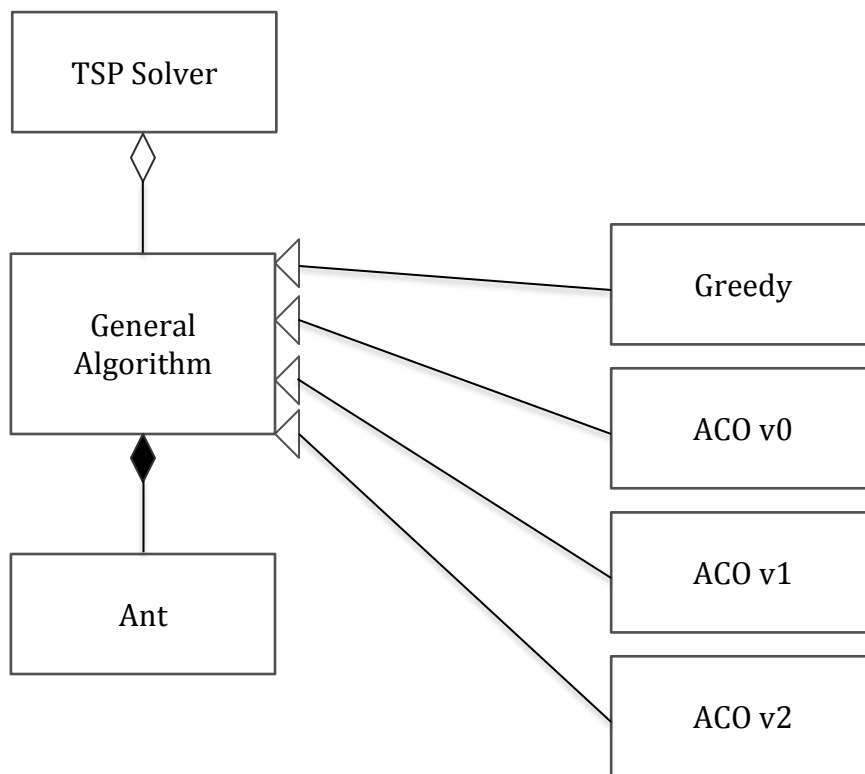
CLASS: Ant	
Responsibilities	Collaborators
This class is responsible for the implementation of an “artificial ant”.	Algorithm

Class Diagram

The Class diagram in the Use-case design is a simple representation of the relationship existing between all the classes.

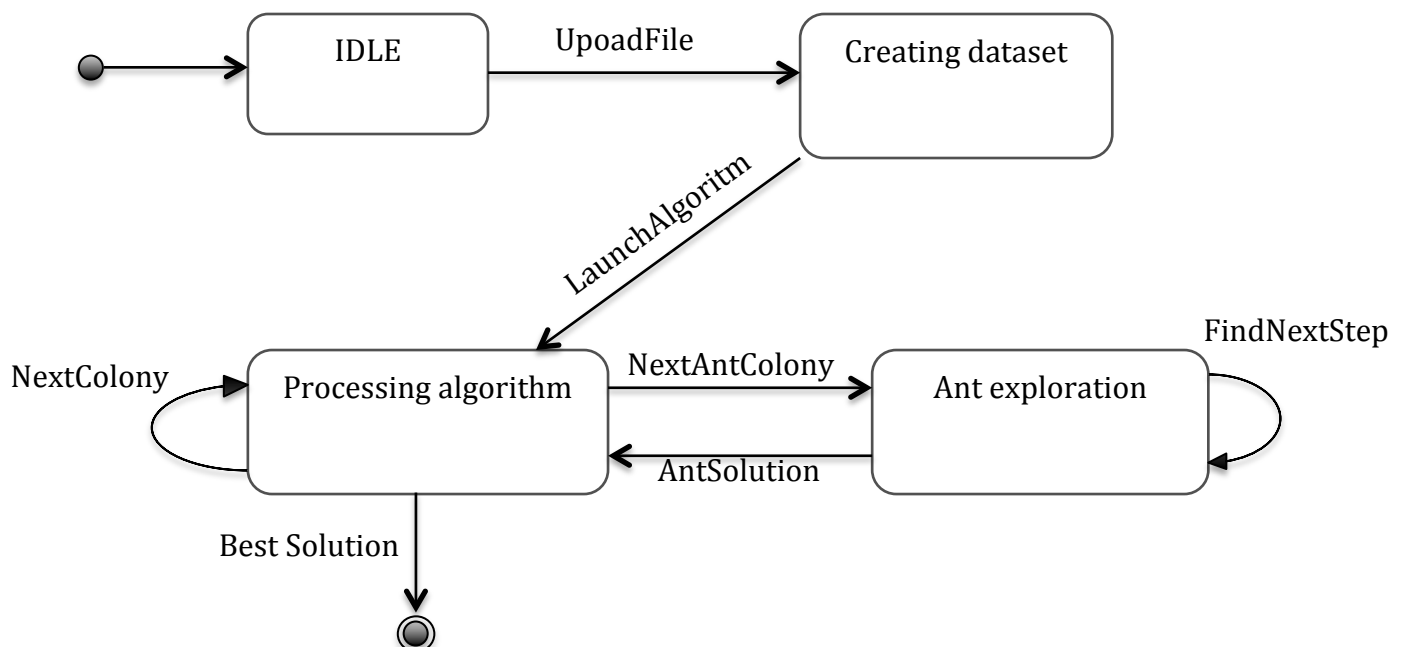
It doesn't show any information about the classes at this stage, but it helps for going throw the Analysis Model stage.





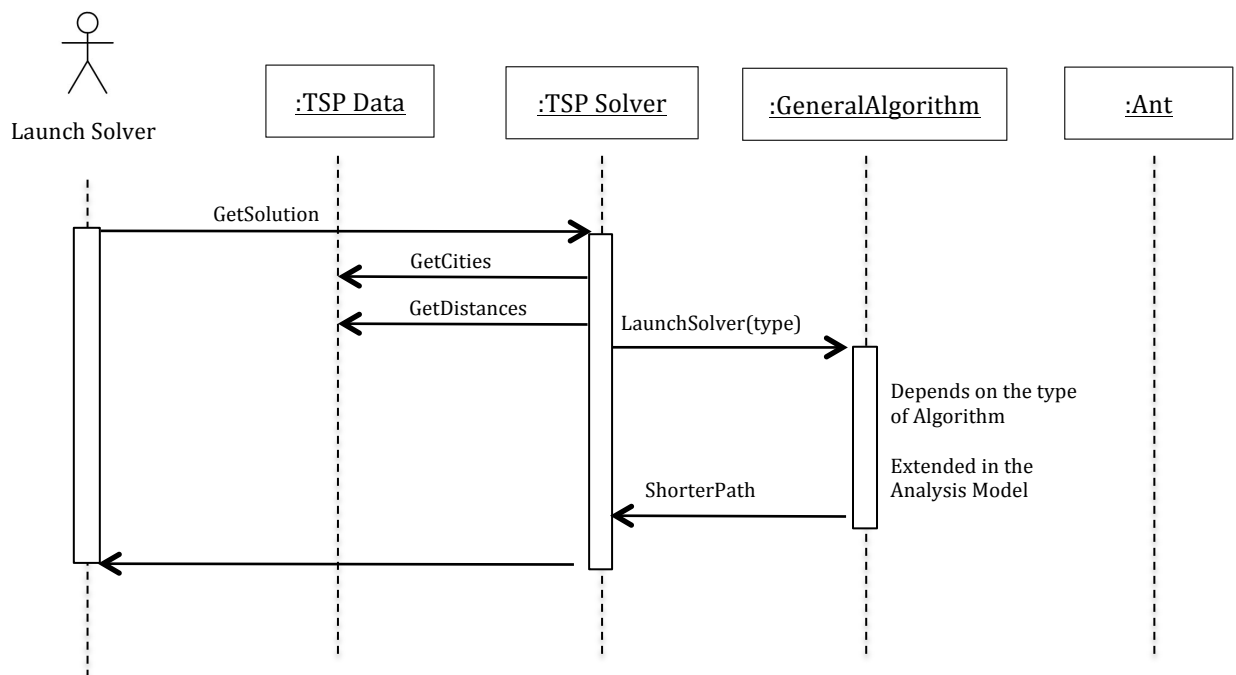
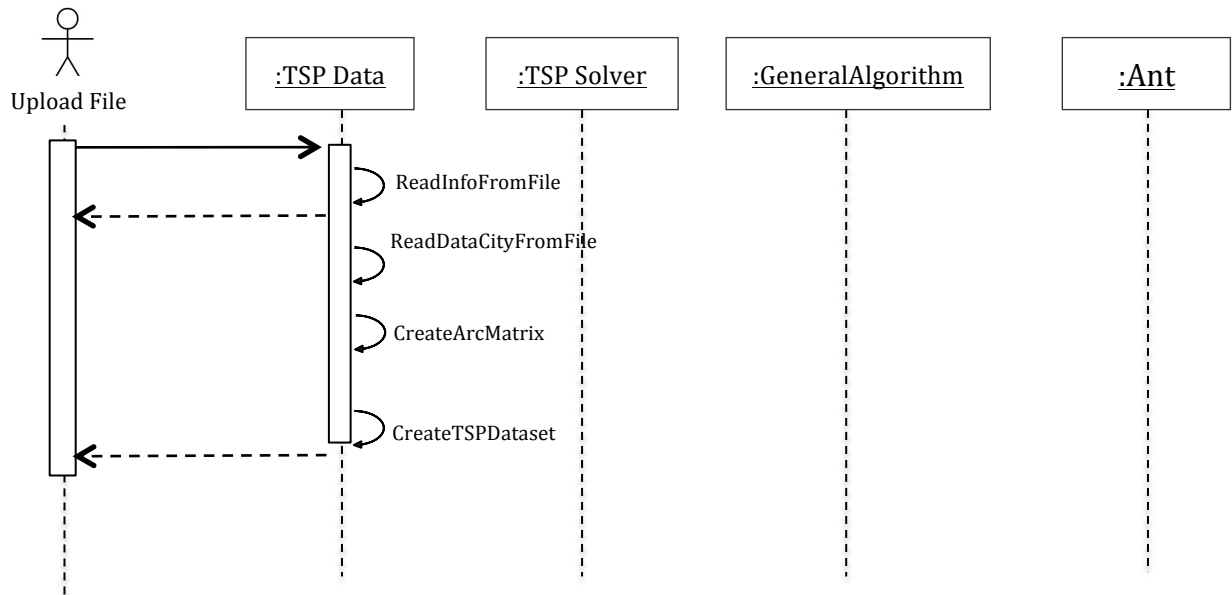
Statechart Diagram

In this design, we can define four principal states.
 The initial state is the IDLE, where the program waits for data to be uploaded.
 Once it happens, it creates a dataset and it waits until the algorithm is launched.
 The Algorithm process can be divided into two states, the main process and the ant exploration process.



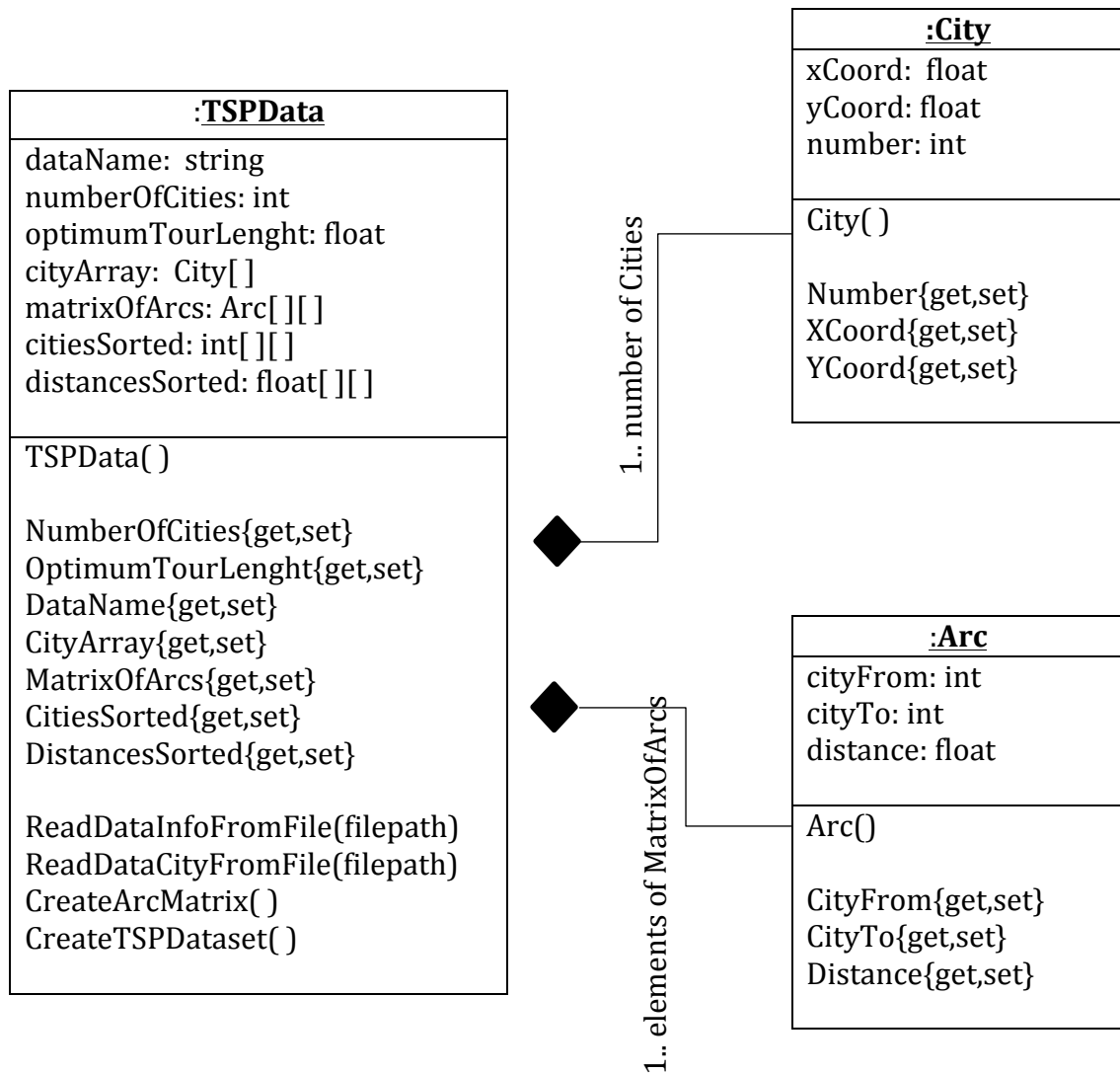
Interaction Diagram

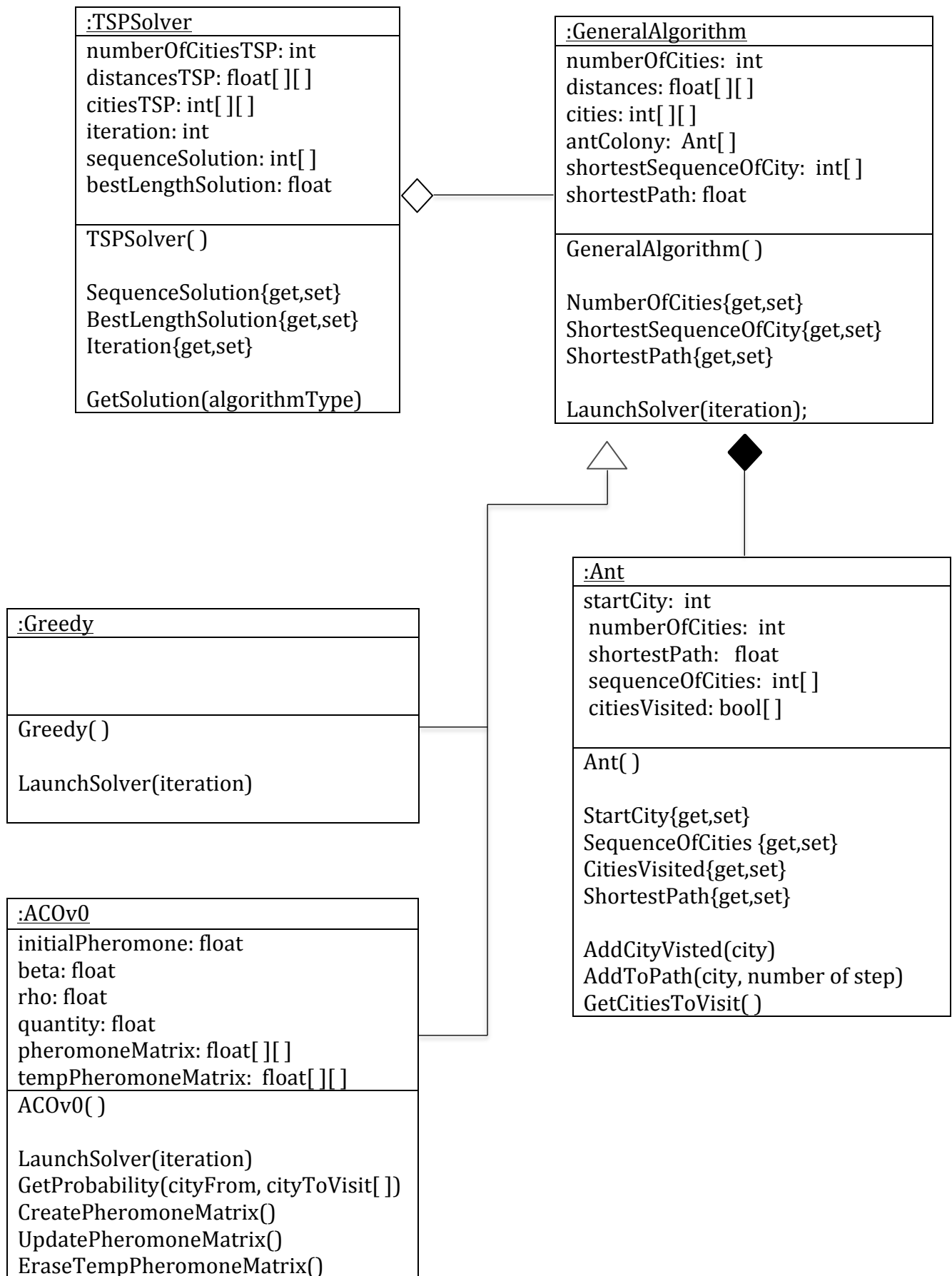
The Interaction Diagram is responsible for showing the flow of the system. In this case, it's been considered worthwhile to use the Sequence Diagram as type of Interaction Diagram.



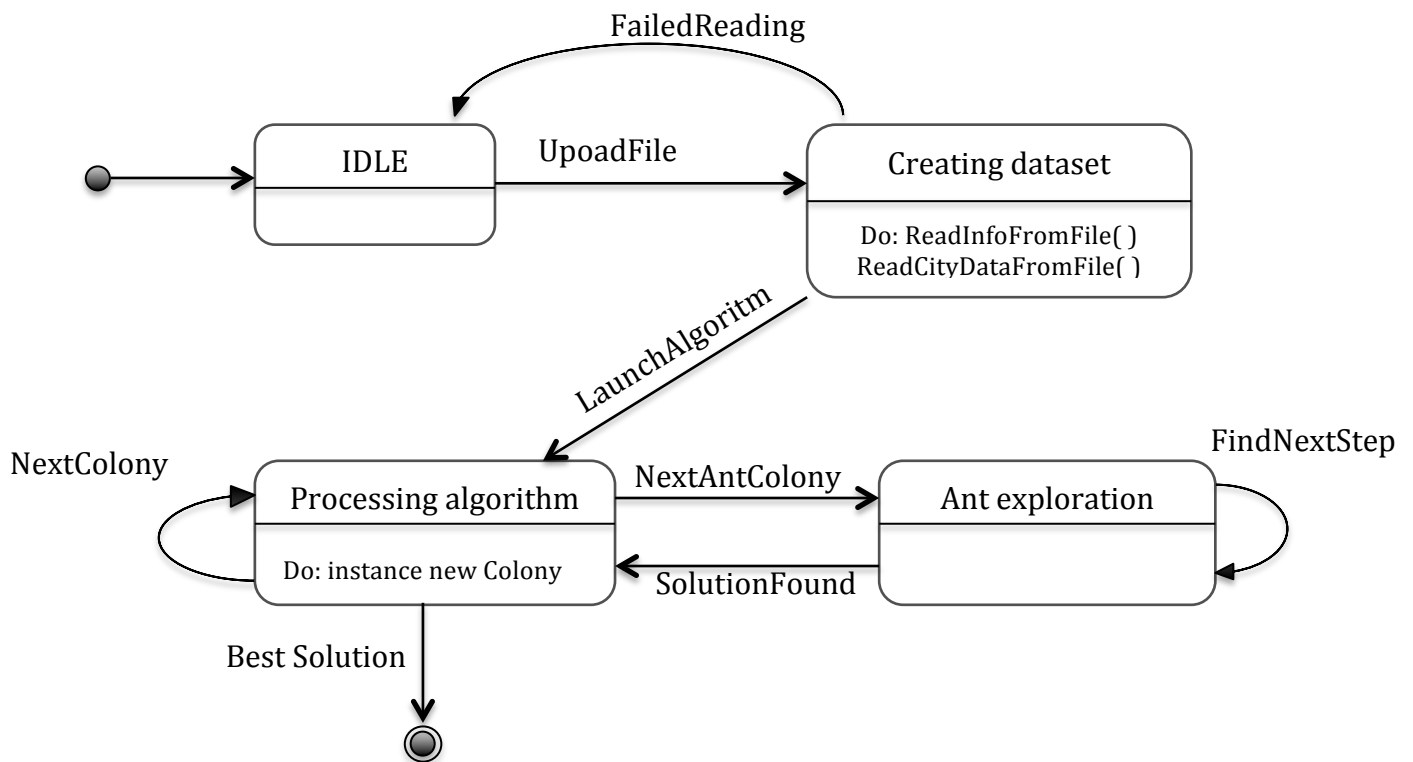
Analysis Model

Class diagram

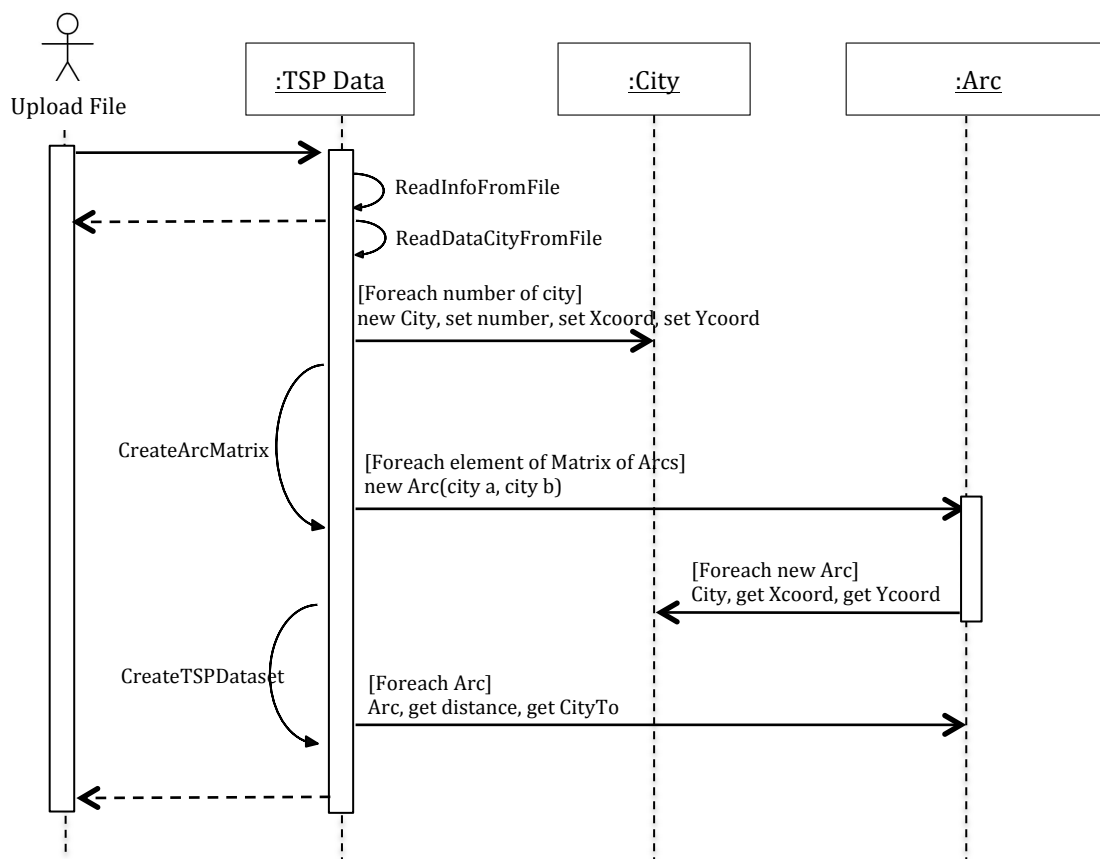




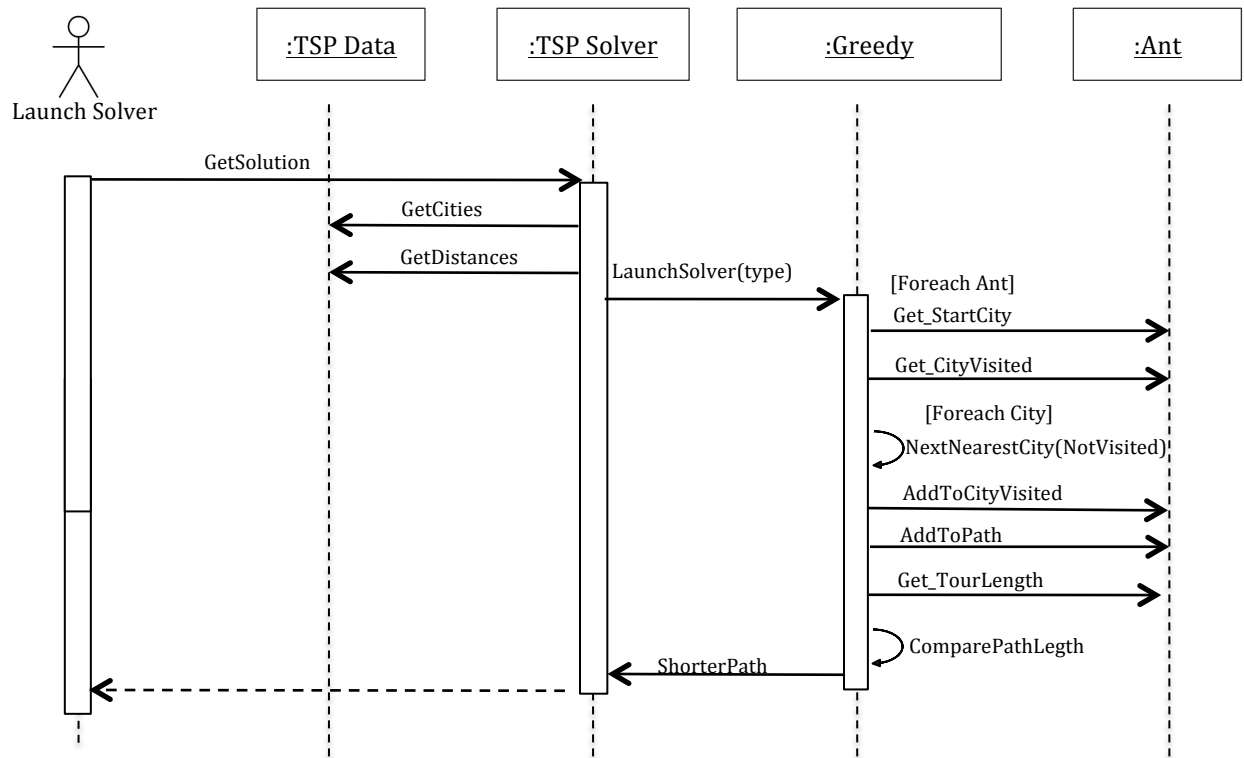
Statechart Diagram



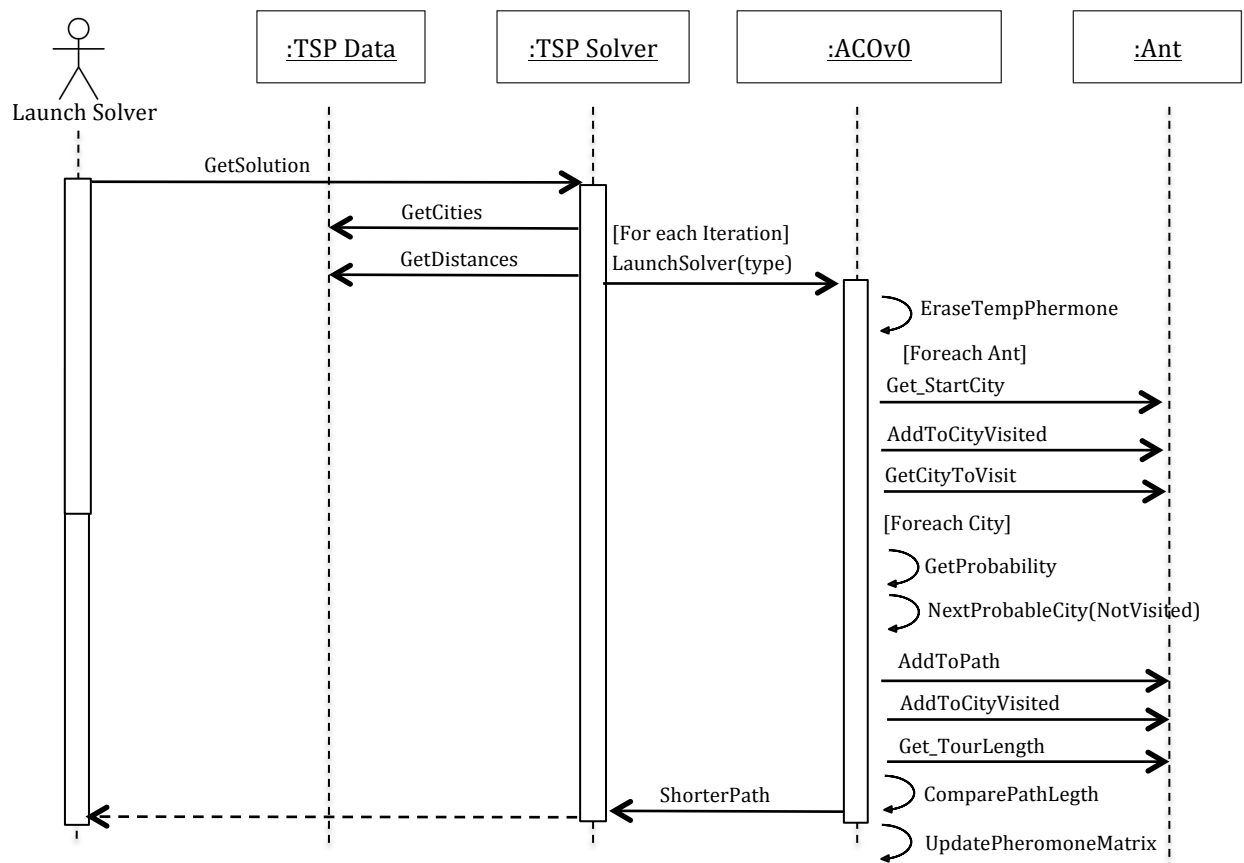
Interaction Diagram



GREEDY ALGORITHM

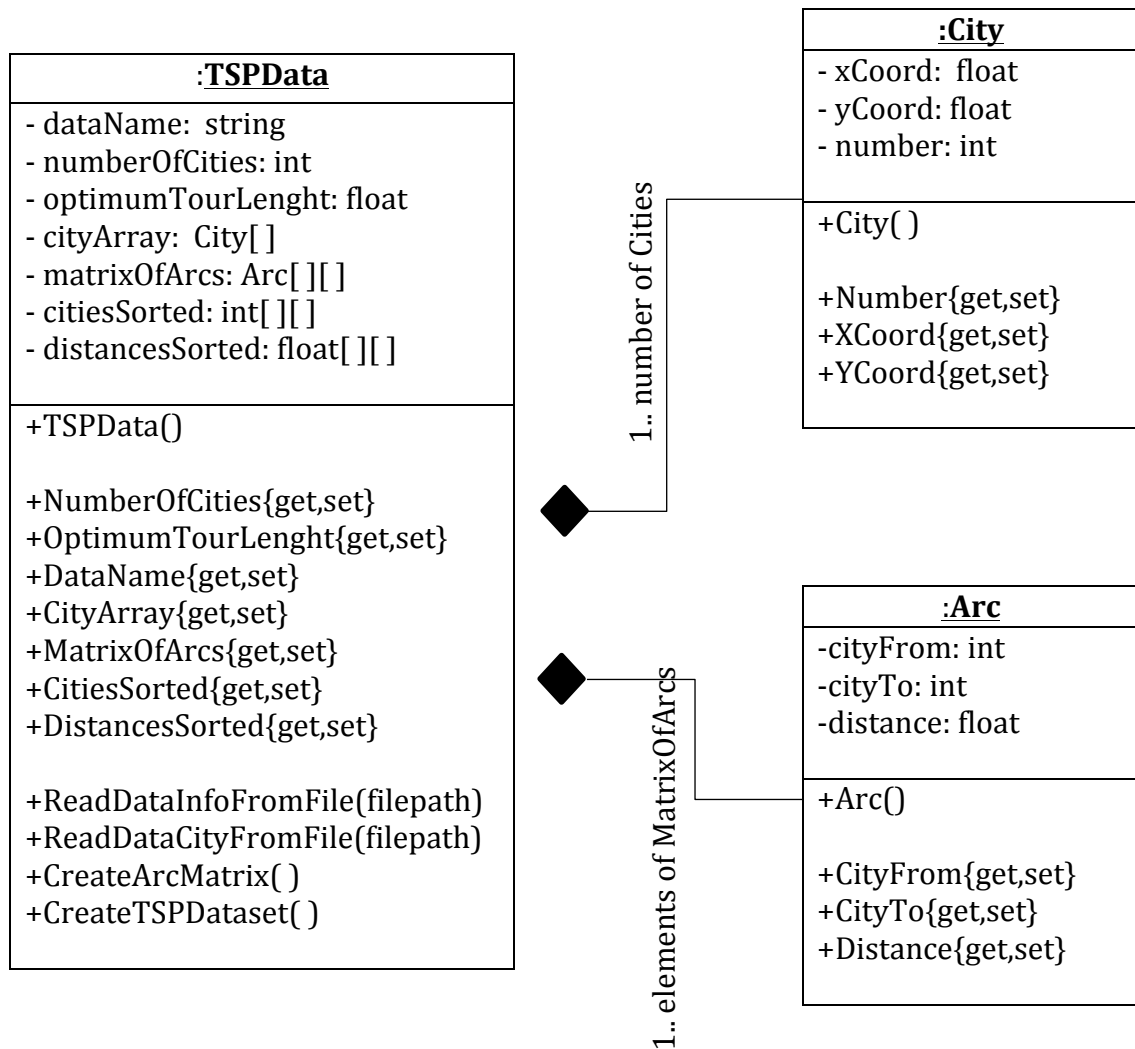


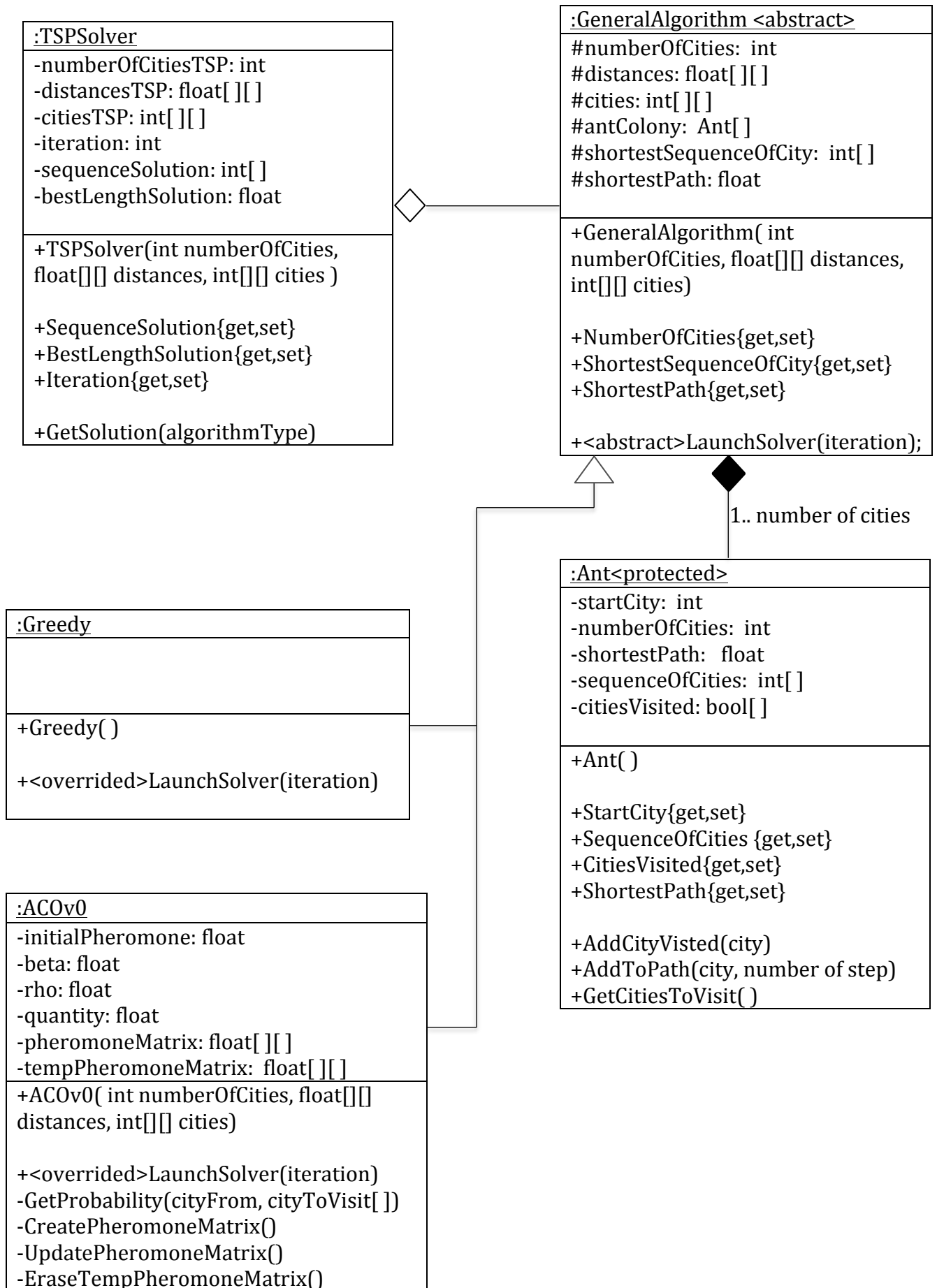
ACO_v0 ALGORITHM



Design Model

Class diagram





Details of Algorithm implementation

The matrices produced by the *TSPData* class used in the *TSPSolver* class have these characteristics:

- Each Row in the matrix of Distances represents the origin city.
- Each Element represents the distance to the city of destination.
- The city of destination is the relative element in the matrix of Cities (the element with same indexes).

The reason behind this division is due to the pre-sorting process applied to the data structure. This choice was taken in order to reduce the complexity of the data structure during the computation.

However, this has been contradicted by the results.

Despite *Greedy* is really optimized, the *ACO algorithms* are not; a different implementation with better results has been tested but not implemented on this project.

Greedy Algorithm

With the matrices already sorted, the next city is always the next element in the matrix of destination cities that represent a city not been visited yet.

In this case, the matrix of distances is used just for retrieve the value of the distance. In fact the choice of 'next city' doesn't involve operations on the matrix of distances.

In addition, being *Greedy* a deterministic algorithm it doesn't need to be iterated for more than one colony.

Pseudo-code:

```
// Iteration for all the ants in the colony
For all the Ants in the colony
Begin
    Ant.AddCityVisted(StartCity)
    presentCity = Ant.StartCity

    //Iteration for all the cities of the TPS problem
    For all the cities
    Begin
        NextCityFound = false;
        if (Ant.CitiesVisited[cities[presentCity][k]] == false && NextCityFound == false)
            distanceToNextCity = distances[presentCity][k]
            numberOfNextCity = this.cities[presentCity][k]
            Ant.AddCityVisted(numberOfNextCity)
            Ant.AddToPath(numberOfNextCity, numberOfIteration)
            presentCity = numberOfNextCity
            pathLenght = pathLenght + distanceToNextCity
            NextCityFound = true
    End
```



```

Ant.Path = pathLenght
if (ShortestPath == 0)
    ShortestPath = Ant.Path
    ShortestSequenceOfCity = Ant.SequenceOfCities

else if (Ant.Path < ShortestPath)
    ShortestPath = Ant.Path
    ShortestSequenceOfCity = Ant.SequenceOfCities
End

```

ACO v0 algorithm

In the ACOv0, a pheromone deposit and evaporation process was added as well as a decision for the next city based on a local probability value.

Unlike the Greedy Algorithm, all the ACO Algorithms need to be iterated so that the pheromone event can interact with the search for the shortest tour length of each ant.

Pseudo-code:

```

//Iteration of the ACO Algorithm
Erase(TemporaryPheromoneMatrix)

//Iteration for all the ants in the colony
For all the Ants in the colony
Begin
    Ant.AddCityVisted(StartCity)
    presentCity = Ant.StartCity

    //Iteration for all the cities of the TPS problem
    For all the cities
    Begin
        Ant.GetCitiesToVisit()
        Probabilities[] = GetProbability(presentCity, CitiesToVisit)

        MaxProbable = Max.Probabilities[]
        RetriveIndex(MaxProbable)

        distanceToNextCity = distances[presentCity][MaxProbable]
        numberOfNextCity = this.cities[presentCity][MaxProbable]

        AddPheromone(TemporaryPheromoneMatrix)

        Ant.AddCityVisted(numberOfNextCity)
        Ant.AddToPath(numberOfNextCity, numberOfIteration)

        presentCity = numberOfNextCity
        pathLenght = pathLenght + distanceToNextCity

    End

    Ant.Path = pathLenght;
    if (ShortestPath == 0)
        ShortestPath = Ant.Path
        ShortestSequenceOfCity = Ant.SequenceOfCities

    else if (Ant.Path < ShortestPath)
        ShortestPath = Ant.Path
        ShortestSequenceOfCity = Ant.SequenceOfCities
    End
UpdatePheromone()

```

ACO v1 algorithm

In the ACOv1 algorithm, the process of pheromone deposit was modified using a secondary matrix for keeping track of the pheromone deposited by the ant that made the shortest tour length inside a colony.

Unlike the previous pheromone matrix that is erased every time a new ant colony is created, this new matrix is constantly erased for each new ant.

(Parts of the code added to the previous version have been highlighted)

```
//Iteration of the ACO Algorithm
Erase(TemporaryPheromoneMatrix)

//Iteration for all the ants in the colony
For all the Ants in the colony
Begin
    Ant.AddCityVisted(StartCity)
    presentCity = Ant.StartCity

    //Iteration for all the cities of the TPS problem
    For all the cities
    Begin
        Ant.GetCitiesToVisit()
        Probabilities[] = GetProbability(presentCity, CitiesToVisit)

        MaxProbable = Max.Probabilities[]
        RetriveIndex(MaxProbable)

        distanceToNextCity = distances[presentCity][MaxProbable]
        numberOfNextCity = cities[presentCity][MaxProbable]

        AddPheromone(TemporaryPheromoneMatrix)
        AddPheromone(TemporaryPheromoneMatrix_BestTourLength)

        Ant.AddCityVisted(numberOfNextCity)
        Ant.AddToPath(numberOfNextCity, numberOfIteration)

        presentCity = numberOfNextCity
        pathLenght = pathLenght + distanceToNextCity

    End

End

Ant.Path = pathLenght;
if (ShortestPath == 0)
    ShortestPath = Ant.Path;
    ShortestSequenceOfCity = Ant.SequenceOfCities
    PheromoneMatrix_BestTourLength = TemporaryPheromoneMatrix_BestTourLength
else if (Ant.Path < ShortestPath)
    ShortestPath = Ant.Path
    ShortestSequenceOfCity = Ant.SequenceOfCities
    PheromoneMatrix_BestTourLength = TemporaryPheromoneMatrix_BestTourLength
else
    Erase(TemporaryPheromoneMatrix_BestTourLength)
End
UpdatePheromone()
```

ACO v2 algorithm

In the ACOv2 algorithm the decision related to the 'next city' was modified and a 'pseudo-random' process added.

In this case, a threshold verify the weight of the 'max probable' next city, if it is under a certain level, than the decision of the next city is transferred to a random process that select one of the 'most probable' next cities.

(Parts of the code added to ACOv0 have been highlighted)

```
//Iteration of the ACO Algorithm
Erase(TemporaryPheromoneMatrix)

//Iteration for all the ants in the colony
For all the Ants in the colony
Begin
    Ant.AddCityVisted(StartCity)
    presentCity = Ant.StartCity

    //Iteration for all the cities of the TPS problem
    For all the cities
    Begin
        Ant.GetCitiesToVisit()
        Probabilities[] = GetProbability(presentCity, CitiesToVisit)

        MaxProbable = Max.Probabilities[]

        //Pseudo-Random Probabilistic decision
        If (MaxProbable > ProbabilityThreshold)
            RetriveIndex(MaxProbable)
            distanceToNextCity = distances[presentCity][MaxProbable]
            numberOfNextCity = cities[presentCity][MaxProbable]

            Else if(Probabilities.Length > threshold)
                Sort.Probabilities
                GenerateRandomIndex(from 0 to threashold)

                distanceToNextCity = distances[presentCity][RandomIndex]
                numberOfNextCity = cities[presentCity][RandomIndex]

            Else()
                RetriveIndex(MaxProbable)
                distanceToNextCity = distances[presentCity][MaxProbable]
                numberOfNextCity = cities[presentCity][MaxProbable]

        AddPheromone(TemporaryPheromoneMatrix)

        Ant.AddCityVisted(numberOfNextCity)
        Ant.AddToPath(numberOfNextCity, numberOfIteration)

        presentCity = numberOfNextCity
        pathLenght = pathLenght + distanceToNextCity

    End

    Ant.Path = pathLenght
    if (ShortestPath == 0)
        ShortestPath = Ant.Path
        ShortestSequenceOfCity = Ant.SequenceOfCities

    else if (Ant.Path < ShortestPath)
        ShortestPath = Ant.Path
        ShortestSequenceOfCity = Ant.SequenceOfCities
End
UpdatePheromone()
```

Account of Structured Testing

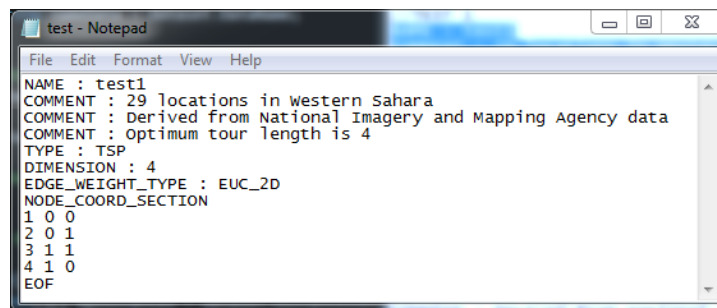
The testing of this system has been approached in a very specific way.

Firs of all, the initial tests were run on a Console Application without the GUI, using the *Console.Write()* and *Console.WriteLine()* methods for printing out the data on the monitor. For this reason, I implemented a set of methods for printing that are not included in the UML design that had been commented and placed at the end of the code of each class.

In order to fully understand the results from these tests, a subset of data entries was created from scratch. Knowing the expected solution, with this method it was possible to validate the functionality of the algorithms implemented.

TSPData Class Testing

Based on using ad-hoc print methods that print the data out on the console monitor, the first test was generated with a really easy dataset. The file has 4 cities, which are placed at the vertexes of a square of length 1.



```
test - Notepad
File Edit Format View Help
NAME : test1
COMMENT : 29 locations in western sahara
COMMENT : Derived from National Imagery and Mapping Agency data
COMMENT : Optimum tour length is 4
TYPE : TSP
DIMENSION : 4
EDGE_WEIGHT_TYPE : EUC_2D
NODE_COORD_SECTION
1 0 0
2 0 1
3 1 1
4 1 0
EOF
```

Figure 1 - Custom set of entries with 4 cities

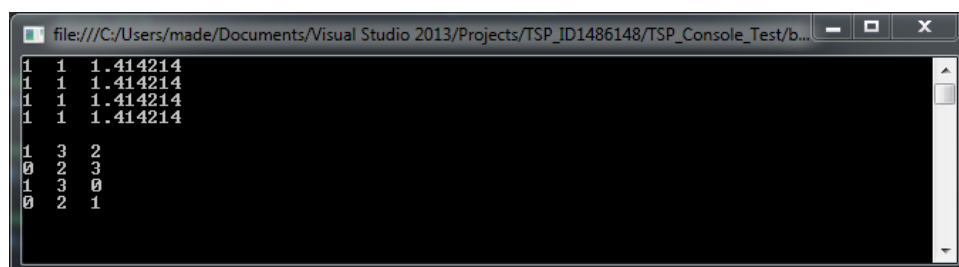
The obtained results were completely congruent with the expectation.

In the first matrix printed out:

- in each row there are all the distances from a starting point (where the row represents the starting point)
- the distances are sorted from the smallest value to the biggest
- there are 12 values, which are all the arcs linking a set of 4 points

In the second matrix printed out:

- all the elements represent the arriving point (and the row represent the starting point)
- the elements in each row are the points excluding the one representing the Row
- there are 12 values, which are all the arcs linking a set of 4 points



```
file:///C:/Users/made/Documents/Visual Studio 2013/Projects/TSP_ID1486148/TSP_Console_Test/b...
1 1 1.414214
1 1 1.414214
1 1 1.414214
1 1 1.414214
1 3 2
0 2 3
1 3 0
0 2 1
```

Figure 2 - Console application results for TSPData class

Those results prove that the TSPData acquisition method from a human readable file works as expected, so I tested directly with the TSP data given for the assignment without encounter any problem.

TSPSolver Class Testing

The test on the TSPSolver class wasn't possible to be achieved with a too easy and symmetric example as the 'square' used for the TSPData class, because a wrong behaviour could have led to a right result.

So, I realised another subset of entries with 7 Points placed as shown in figure.

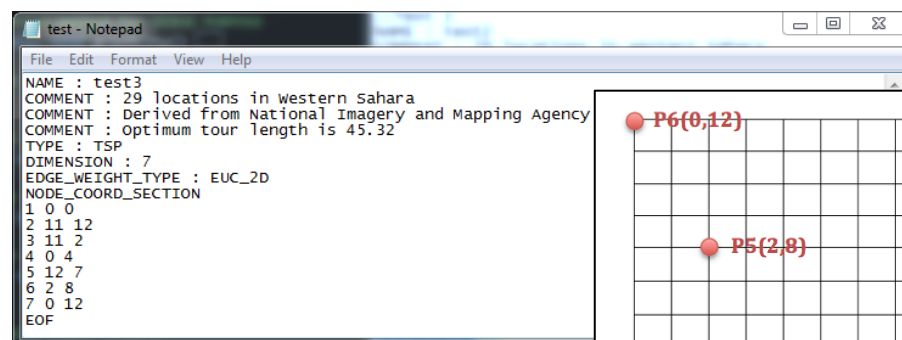


Figure 3 - Custom set of entries with 7 cities

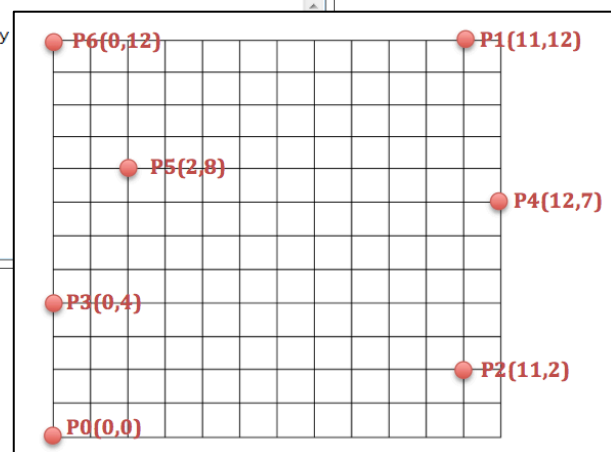


Figure 4 - Graphical view of the custom set with 7 cities

The TSPSolver used the sorted matrices for the distances and for the cities, which was calculated by the TSPData class and verified with the results calculated by hand.

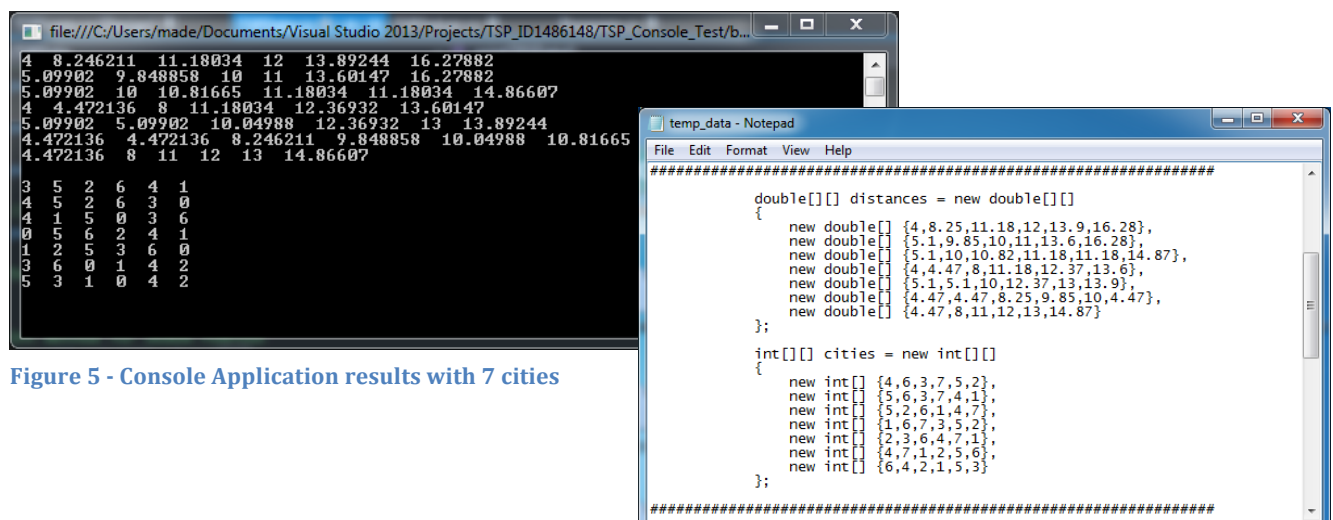


Figure 5 - Console Application results with 7 cities

In this example, the dataset is not symmetric so it was possible to verify the correct behaviour of the Greedy Algorithm.

In fact, starting from a point and making a complete tour choosing the nearest next point as rule return a different result based on which is the first point chosen.

The optimum tour result calculated by hand was 45,32 , which corresponds with the result returned by the computation of Greedy Algorithm.

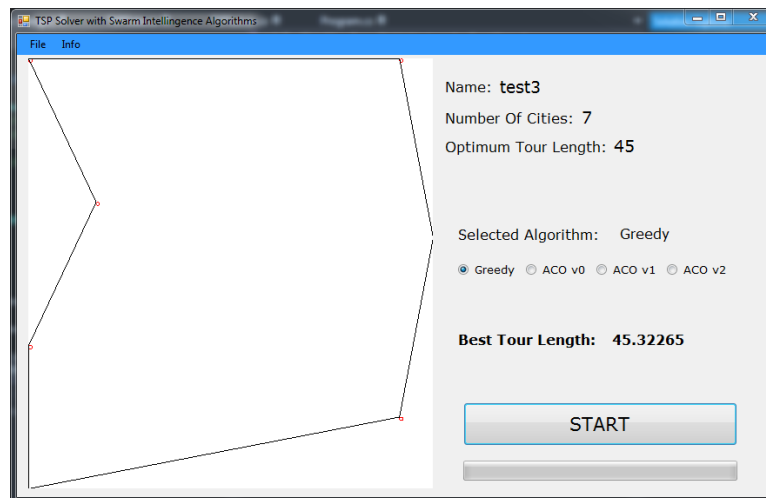


Figure 6 - Windows From - Greedy - 7 cities

For the ACOv0 algorithm a great part of the testing was based on studying the 'update and evaporation process'.

This was achieved using the print methods implemented and verified with the expected behaviour for the custom dataset of 7 cities.

Anyway, the correct behaviour of the ACO couldn't be studied with a data entry that can lead to an Optimum solution with *Greedy algorithm*. Thus, it was mandatory to use a quite complex one, so the 29 cities example had been chosen as the most appropriate (in term of computation time and rapid re-testing).

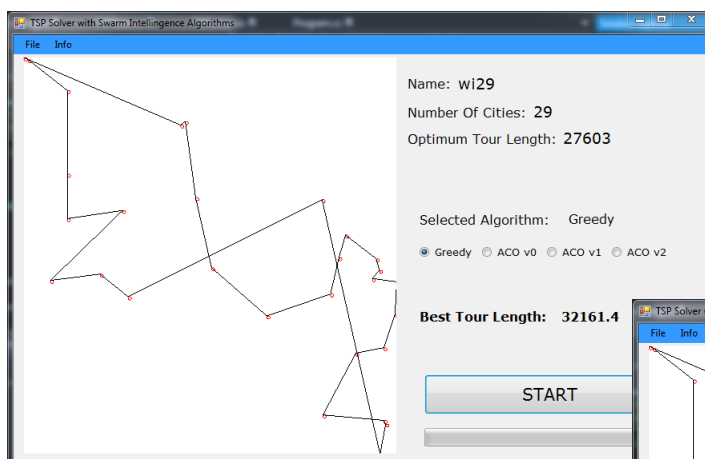


Figure 7 - Windows Form - Greedy - 29 cities

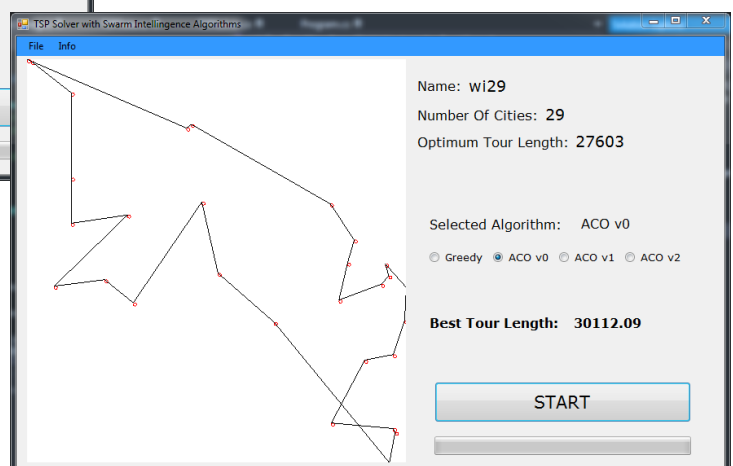
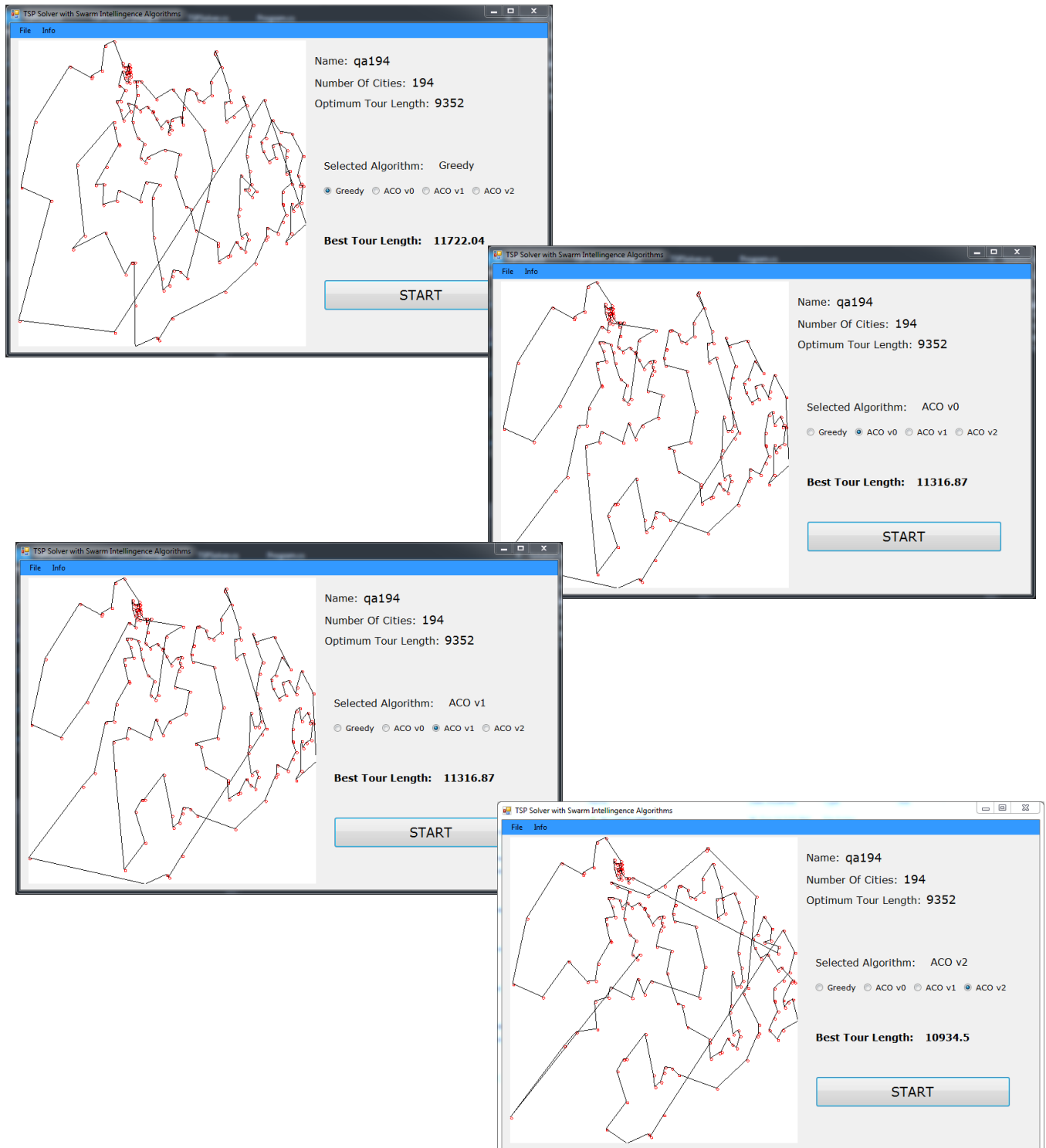


Figure 8 - Windows Form - ACOv0 - 29 cities

Once verified that the ACOv0 returned better results than Greedy, a series of test have been run for tuning the parameters of this Algorithm.

The ACOv1 and ACOv2 were then implemented and tested on a set of data entries with more than 100 cities. Unfortunately, the ACOv1 has never return better results than ACOv0.



Conclusions

The results obtained for this implementation are sufficiently satisfactory for what concerns the functional requirements of this design.

The Greedy algorithm, as the ACOv0, return results that in the expected range of accuracy. The ACOv1 and ACOv2 are not been tested sufficiently on large data entries in order to tune the parameters properly.

Certainly, the main improvements to be done are addressed to the data structure for the TSP solving algorithms. The data structure should be conceived in order to minimize the operations required to select the next City in the ACO, such as getting the Local Probability and updating the Pheromone.

Also, the approach to OOP used in this project is definitely to enhance. Even though the UML helped to understand the methodology for building a design in an Object Oriented Language, the process of organizing the code for such technical and 'domain' based problems needs to be exercised.

Regarding the quality of the ACO algorithms, it could be interesting to implement a strong approach to multithreading based on smaller sub-systems of the exploration space. Selecting multiple sets of near cities and treating each of them as a smaller TSP dataset. Thus the pheromone update will confine the exploration space on an optimized subset of suitable 'next city'.

This assignment was a great opportunity to improve the knowledge on OOP Languages. Although the results obtained are not completely fulfilled for all the requirements of this system, I acquired a good understanding of complex algorithm efficiency and the relationship between them and their related data structure.