School of Electronic, Electrical and Systems engineering

UNIVERSITY OF BIRMINGHAM

MSc Embedded Systems

Object Oriented Programming using C#

Swarm Intelligence

Luciano Juarez Rivera ID 1487897

Professors: Dr. Mike Spann, Mr. David Pycock

1/12/2014

Introduction

Since early times, humans have developed technology based on nature observation. By reproducing ingenious natural behavior, potential solutions to complex problems have been generated. One of these behaviors is called *Swarm Intelligence*. Swarm intelligence takes the dynamic interaction between animal species as a basis, which allows them to accomplish significantly complex tasks (Teodorovic, 2008).

In this particular case, the emphasis is on an algorithm called Ant Colony Optimization (ACO), which consists of replicating the strategy ant colonies use to find the best path from the nest to a food source. More formally, "Ant colony optimization is a metaheuristic in which a colony of artificial ants cooperate in finding good solutions to difficult discrete optimization problems" (Dorigo & Stützle, 2004: 33).

Real ants use chemical substances to communicate between them, leaving a trail of pheromone in the path they go through. The higher the pheromone levels in a path, the bigger the number of ants that will go into it. This means that initially every trail is a potential trail to move into. After more ants find a path, the better paths are "traced" by the highest level of pheromones. It is observable that real pheromone evaporates to forget non optimal routes. Analogically, artificial ants will be building partial solutions to a discrete state problem. The same way each ant will leave artificial pheromone, which is local information that depends on the length of the arc from a node to another, and the previous pheromone deposit on that arc. This translates that an ant will be more likely to choose an arc if the distance from node *i* to node *j* is small, and the pheromone deposit on arc *i* to *j* is high. After an ant finishes a tour, it will leave virtual pheromone on the arcs it went through. This way, more efficient routes will be registered on memory. To prevent path repetition, virtual pheromone is also conditioned to "evaporate" with a constant rate. After more solutions are found, higher global quality solutions are generated (Dorigo & Stützle, 2004).

One classic problem of discrete optimization is the Travelling Salesman problem (TSP). "The TSP is the problem of finding a minimum length Hamiltonian circuit of the graph, where a Hamiltonian circuit is a closed walk (a tour) visiting each node of G exactly once" (Dorigo & Stützle, 2004: 40). Where G is a graph with N nodes and A arcs that connects all of the nodes. Therefore, TSP is a problem where a salesman wants to find the shortest path to visit all of the cities in a region, starting from hometown and visiting all of them only once, returning home after that.

In accordance with the specification, the following project implements ACO to solve the TSP.

Use Case View

To tackle the project design, the Unified Modeling Language will be used. The first step is to understand and indentify the user requirements. To help with it, a brain storming is made.

1.1 Brain Storm

Potential requirements:

- Apply different versions of the Ant Colony Optimization (ACO) algorithm to solve the Travelling Salesman Problem (TSP)
- Open and read a file containing the TSP dataset
- Process the file to obtain the necessary information for the algorithms
- Display a map of the cities
- Draw a map of the selected dataset
- Process the algorithm results, and display them in a user friendly and easy to read interface
- Exit and close application at any moment
- Override a current algorithm if a new file is selected while running
- Once an algorithm is selected, the program will automatically generate and display a solution
- Relevant data is processed and displayed when a file is opened

Because this is a pure software application, there is no risk associated with any of these requirements. Therefore, the design will be incorporating all of the mentioned requirements.

Based on the identified requirements and possibilities the user has, one main actor is identified which is the user. The use – case diagram is derived and is as follows:

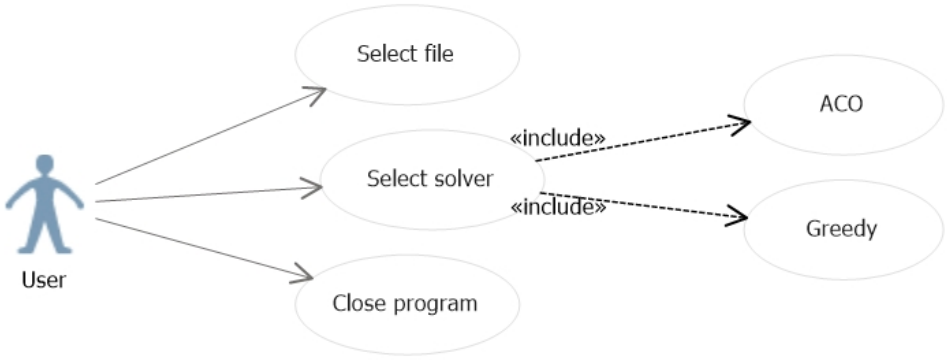


Figure 1 Use-Case Diagram

1.2 Scenario Descriptions

Based on the use- case diagram, a scenario description is made for extracting the system behaviour. This will help to find potential unrecognized states on the use-case diagram, as well as defining the system's main classes.

User: The user chooses a file, by activating the open file menu and selecting a desired file. When the file is selected, its data is read and processed (in order to be prepared for further algorithm selection) and relevant information (file name, dimension, and optimum tour length) is displayed on screen. A map of the dataset is also displayed. When the reading and processing tasks are done, the user selects an algorithm to solve the problem. The selected algorithm executes, and begins solution generation. When the process is finished, the best tour found is displayed, and the path of this tour is plotted on the map. After result displaying, the user can select a new file to solve by following the same initial procedure. If the chooses a file while an algorithm is running, the current algorithm will stop, and the new file preliminary data will be displayed and ready to be solved. If the user selects the close button at any moment, the program will be terminated.

1.3 Class Identification

By analyzing the scenario description, potential classes can be identified properly. Nouns on the description turn into potential classes. From the scenario description before, initial classes are proposed.

Nouns

File, Algorithm, Close button, Screen.

Stereotypical classes

Boundary: screen, close button
Analysis: file, algorithm
Control:

Since there is no immediate identified control class, the User Interface will be the main control and boundary class. It will contain the boundary elements screen and close button as well (these elements are not justified to be a class themselves).

Therefore, the system will be contained within three main classes: GUI, fileReading, Solver.

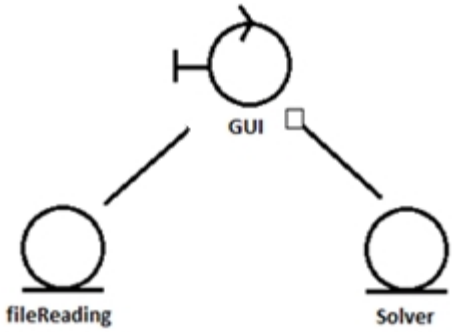


Figure 2 Class Diagram (Stereotypes)

1.4 CRC Cards

Class Responsibility Collaboration (CRC) cards aid on the construction of classes. On each CRC card, the responsibilities of the previous defined classes are set, in order to cover all of the system functional requirements.

Table 1 Class Responsibility Collaboration Cards

Class: fileReading	
Responsibilities	**Collaborators**
The fileReading class is responsible for reading the user selected file, identifying and extracting the information needed by classes Solver and GUI. Information is divided in two main sets which are general information (dimension, name and optimum tour length) and TSP data (city number, coordinates).	GUI

Class: Solver	
Responsibilities	**Collaborators**
The solver class is responsible for processing the information to get it ready for the algorithms. It is responsible to execute the user selected algorithm to solve a particular TSP dataset. After finishing an algorithm, it provides the results (shortest tour found, its length and its path) to GUI class.	fileReading, GUI

Class: GUI	
Responsibilities	**Collaborators**
The GUI class is responsible for updating the screen, whenever fileReading or Solver classes provide information when finishing execution. Updating the screen includes graphing the map and the path, displaying the value of the shortest tour found, as well as the file name, optimum tour length and dimension of the selected dataset. It is responsible for controlling execution.	Solver, fileReading

1.5 Interaction Diagram

Interaction diagrams serve as visual aiders to define class interaction and data exchange between them. The collaboration diagram shows how the classes interact between them. The sequence diagram shows how the user actions are processed within the classes. Collaboration diagram and sequence diagram are shown on the following pages on figure 4 and 5, respectively.

1.6 Statechart Diagram

An analysis of the different states of the system, throws that it contain four main states: Idle, reading file, Algorithm calculating, updating screen. This helps to understand which actions will trigger a change in the system state.

The Statechart is shown on the following pages on figure 3.

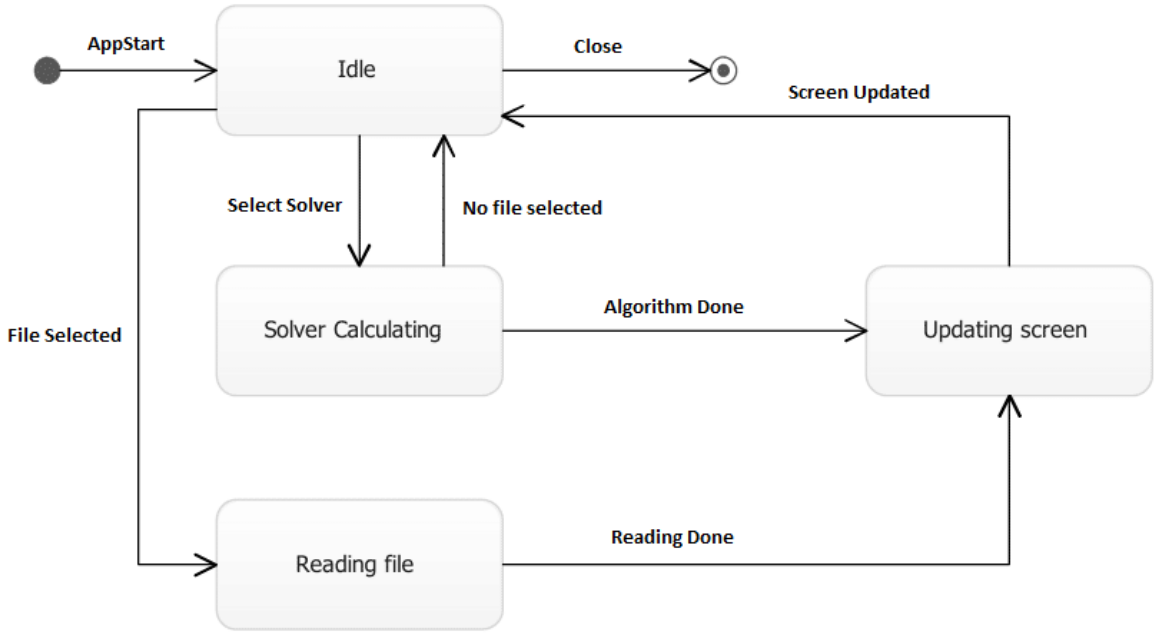


Figure 3 Statechart Diagram

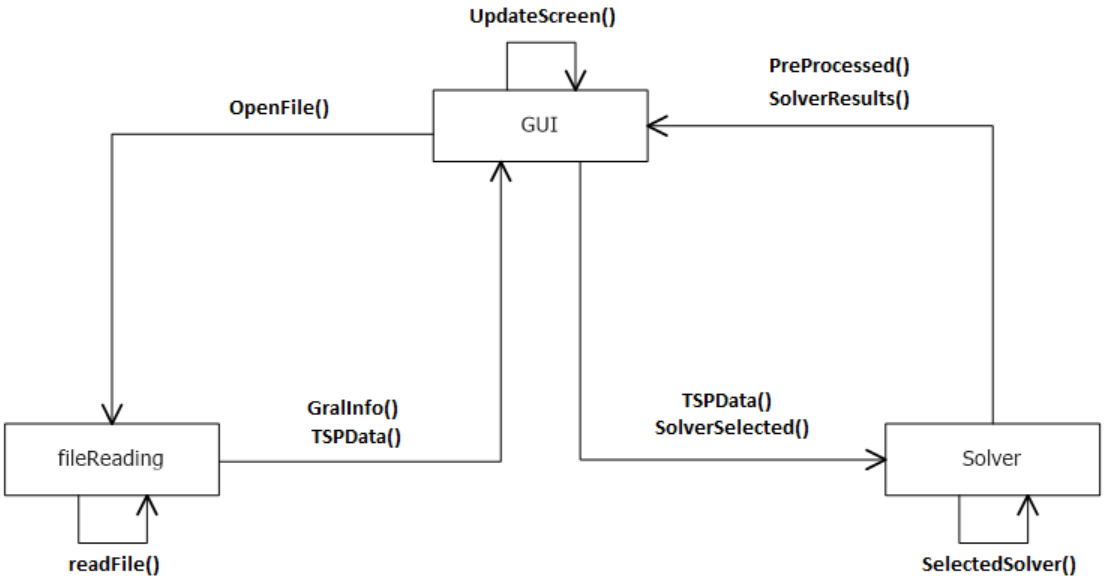
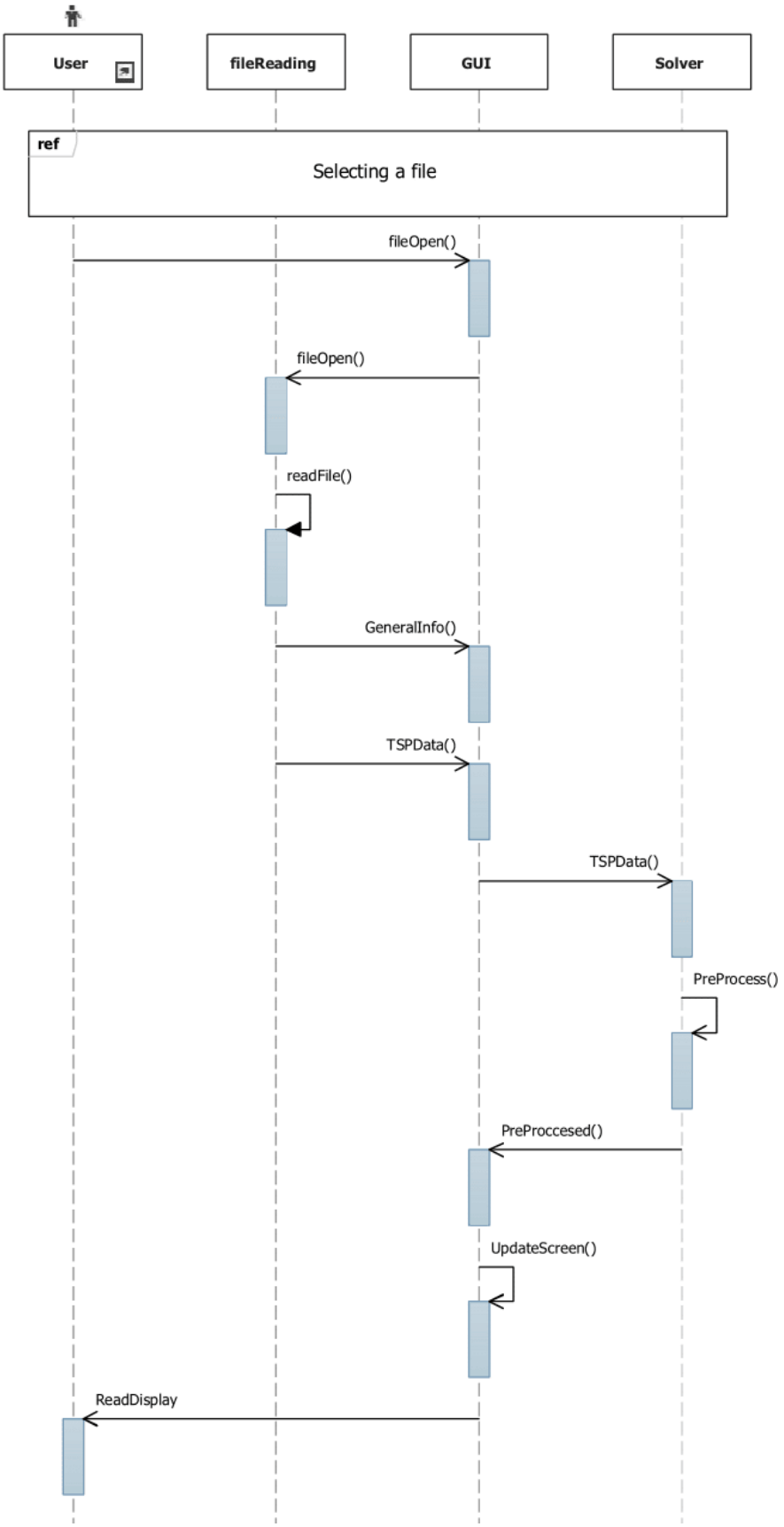


Figure 4 Collaboration Diagram


User
fileReading
GUI
Solver
ref
Selecting a file
fileOpen()
fileOpen()
readFile()
GeneralInfo()
TSPData()
TSPData()
PreProcess()
PreProccesed()
UpdateScreen()
ReadDisplay

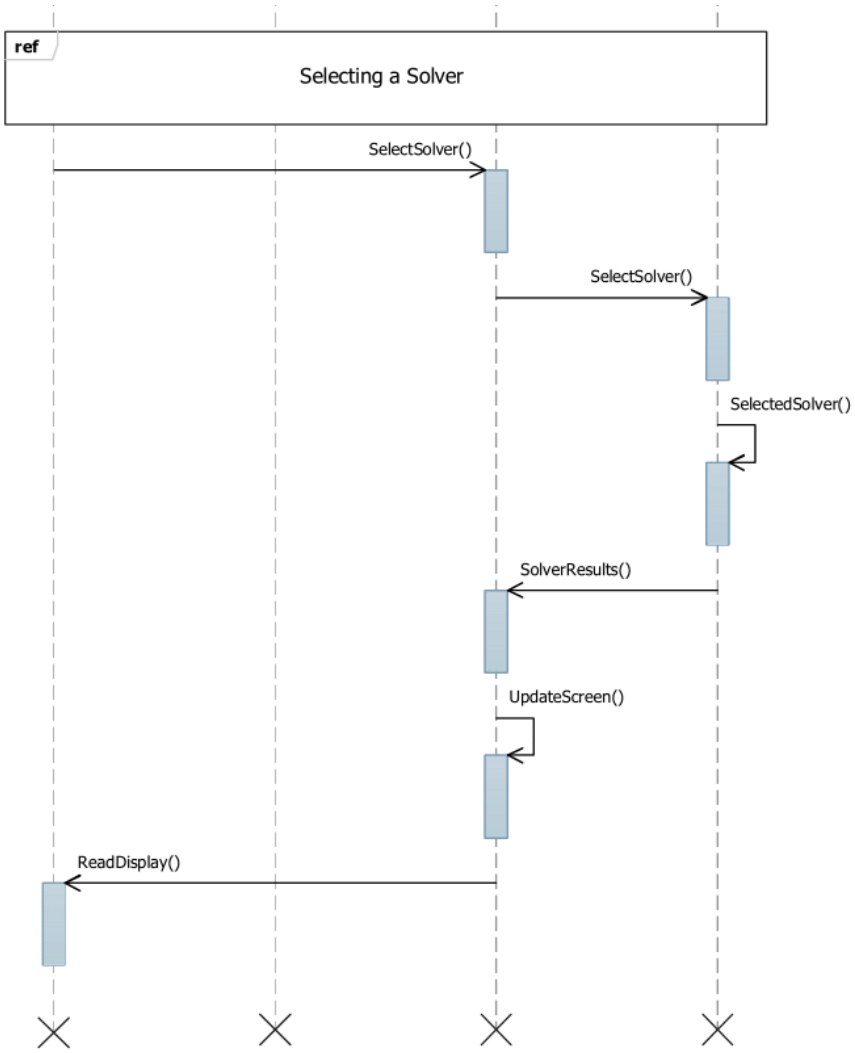


Figure 5 Sequence diagram

Analysis Model

Use case realization gives a general concept of the system functionality. Based on that concept, potential methods and attributes for the classes are identified.

From fileReading Class:

Potential attributes: StreamReader

Potential methods: extractInfo(), readFile(), testData()

From Solver Class:

Potential attributes: Number, Visited, Xcoord, Ycoord

Potential methods: ACOSolver(), CalculateDistance(), GSolver(), RetrieveData()

From GUI Class:

Potential attributes: cities, dimension, distances, highestX, highestY, lowestX, lowest, results, runs, shortestTour, index1, index2, index3, index4

Potential methods: ACOSolver(), greedySolver()

Analyzing different elements such as class diagram, CRC cards, sequence diagram, and refined diagrams, give as a result the actual attributes and methods from the classes.

2.1 Attributes (all private)

Table 2 Attributes

Class	Attribute	Comment
fileReading	StreamReader	File Stream type
Solver	Number	Double
	Visited	Boolean
	Xcoord	Double
	Ycoord	Double
GUI	Cities	Object array
	Dimension	Integer
	HighestX	Double
	HighestY	Double
	LowestX	Double
	LowestY	Double
	Results	Integer Array

Runs	Integer
ShortestTour	Integer
SolverThread	Thread
W	Integer
X	Integer
Y	Integer
Z	Integer

Note that the potential attributes index1 to index4 were changed to letters W to Z, to facilitate coding.

2.2 Methods

Table 3 Methods

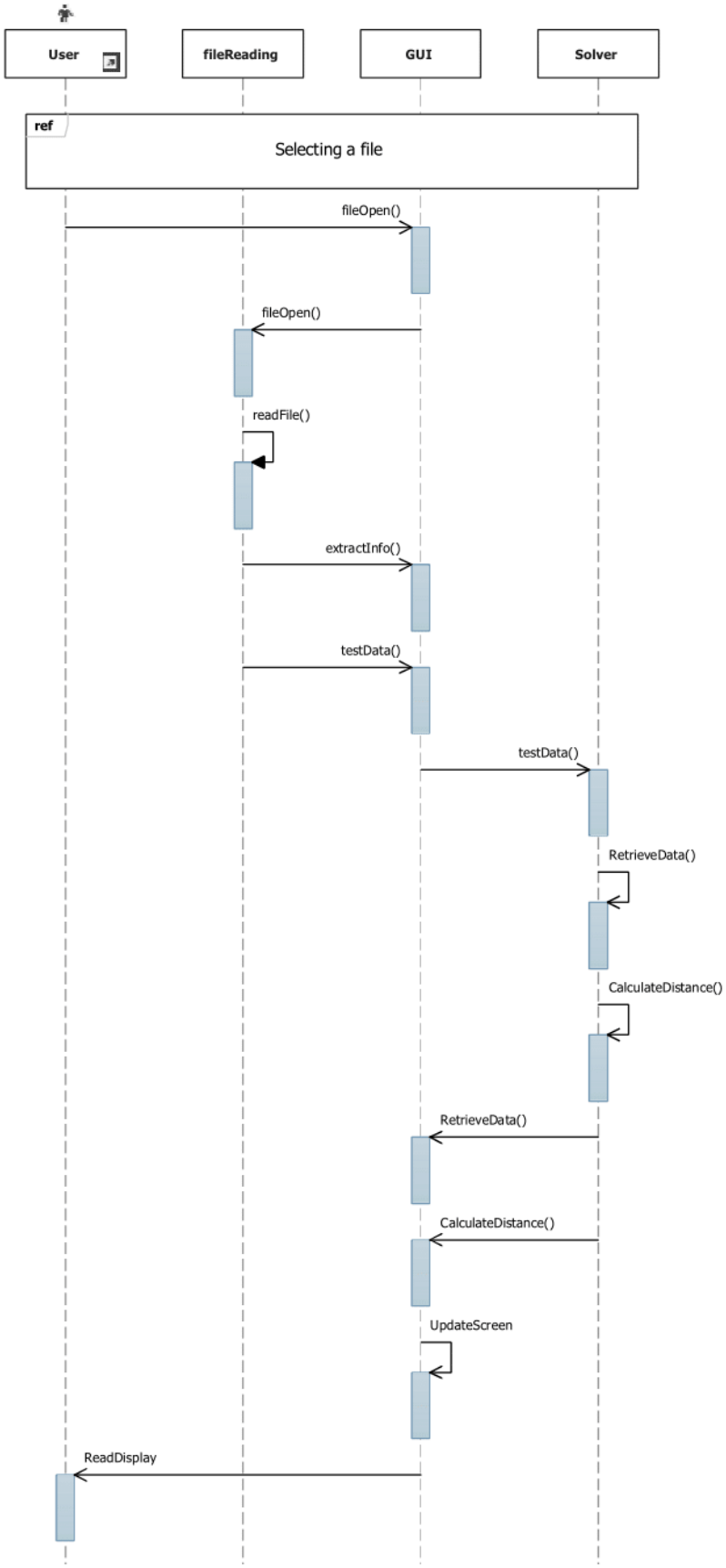
Class	Attribute	Comment
fileReading	extractInfo()	Obtain preliminary relevant information
	readFile()	Read file data
	testData()	Obtain TSP data
Solver	ACOSolver()	
	CalculateDistance()	Pre processing reference table
	GSolver()	
	RetrieveData()	Create cities as objects
GUI	ACOSolver()	Create separate Thread
	greedySolver()	Create separate Thread
	fileOpen()	

Notice that GUI has two methods that will serve as the methods for running the algorithms in different threads of execution.

Based on the new methods, the sequence diagram is revised.

2.3 Sequence Diagram

Revised sequence diagram shown on the following pages on figure 6.


User
fileReading
GUI
Solver
ref
Selecting a file
fileOpen()
fileOpen()
readFile()
extractInfo()
testData()
testData()
RetrieveData()
CalculateDistance()
RetrieveData()
CalculateDistance()
UpdateScreen
ReadDisplay

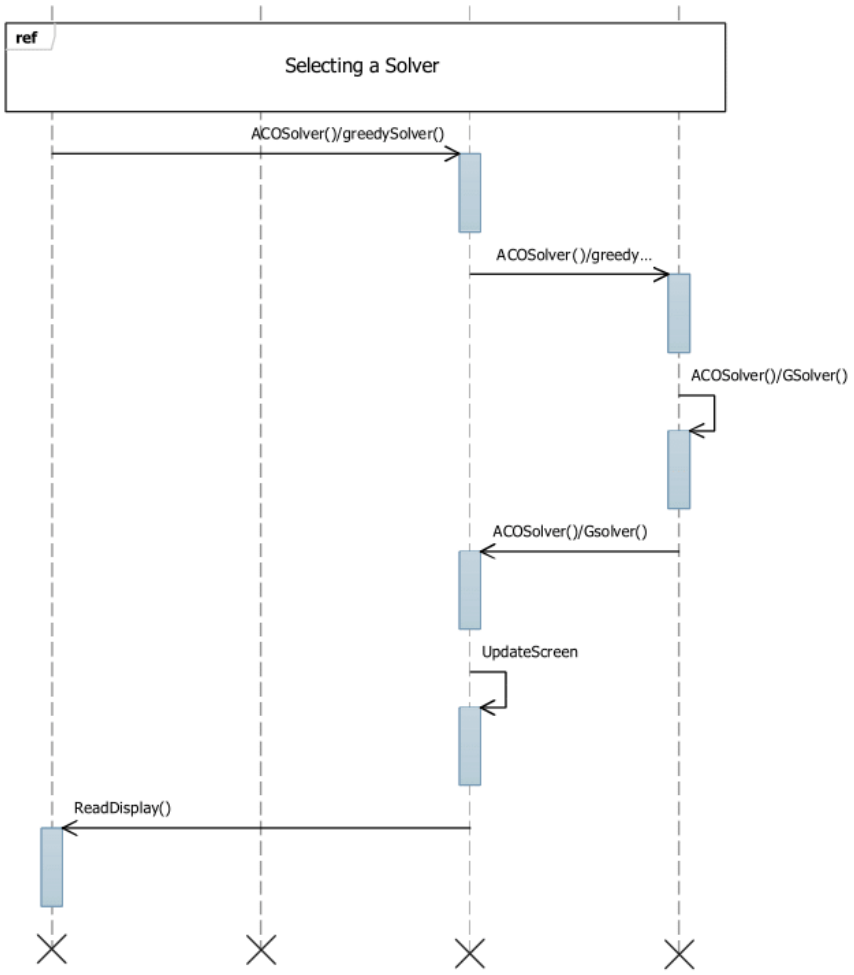


Figure 6 Revised Sequence diagram

2.4 Class Diagram

Now data types are added and methods and attributes are shown. The analysis model class diagram is shown on the next page on figure 7.

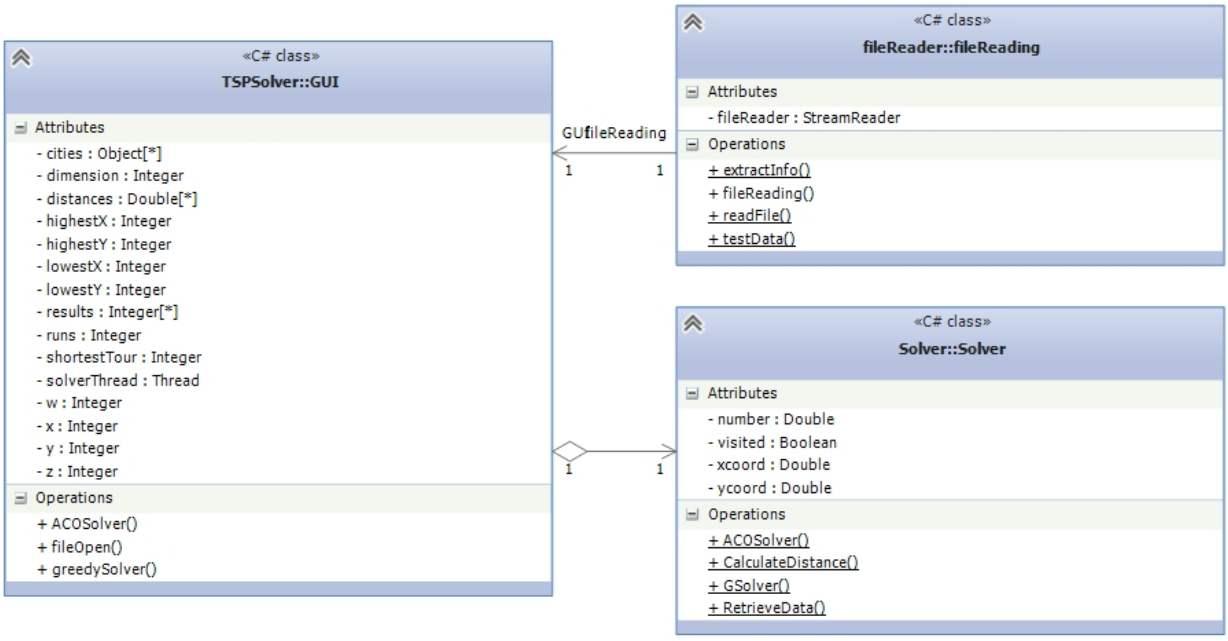


Figure 7 Analysis Class Diagram

It is noticeable in the class diagram that there is no inheritance.

2.5 Statechart Diagram

Analyzing class interaction as well as system behavior reveals that no changes are needed in the use case state chart diagram.

2.6 Non - functional requirements

This software application does not have relevant non – functional requirements.

2.7 Packages

This system design does not require any packages, because the system has only three classes.

Design Modelling

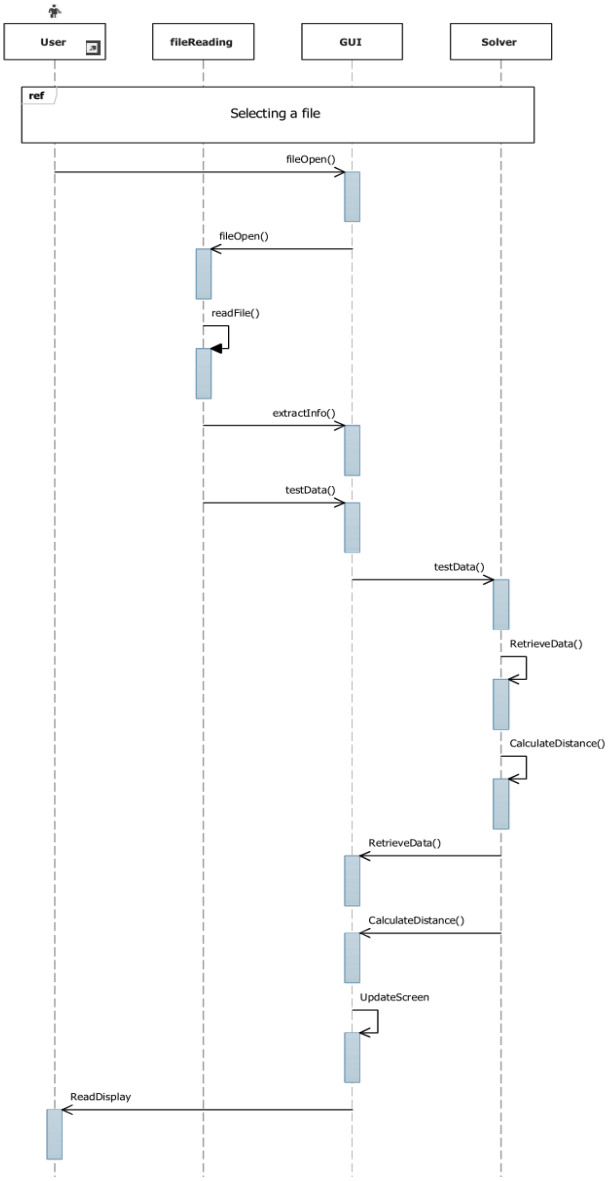
This phase helps to identify the meet of requirements, as well as tuning or improving slight changes.

3.1 Revisit Use – Case Model

The design meets the requirements

3.2 Sequence Diagram

The revised sequence diagram is shown in figure 8



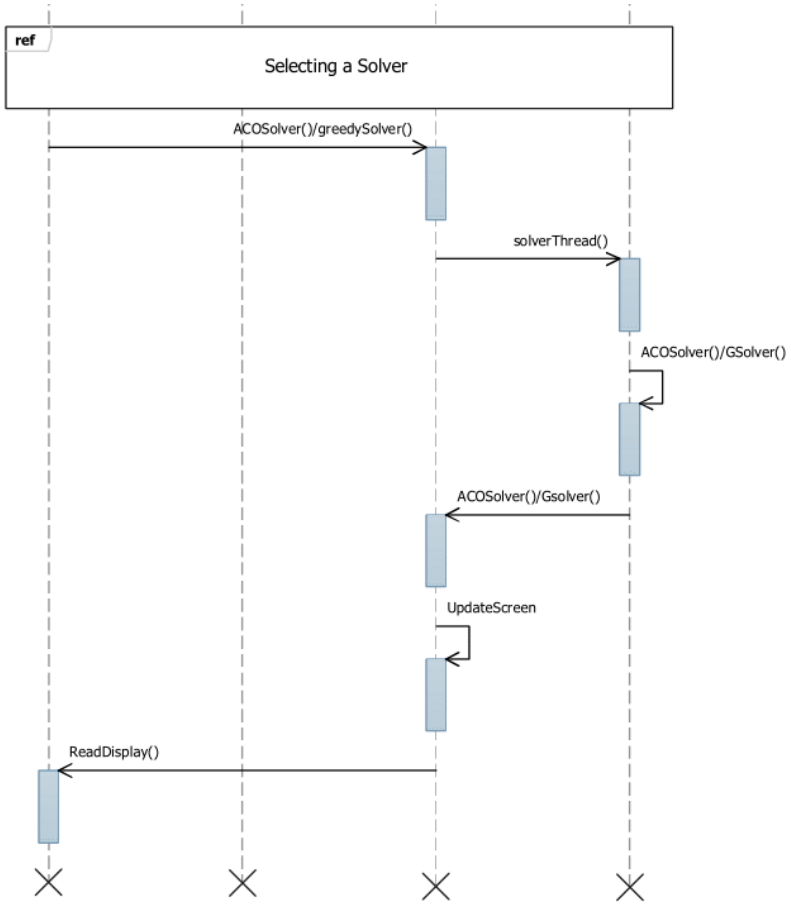


Figure 8 Design model Sequence Diagram

Note that the change was made only at the Selecting a Solver event. Therefore it is shown bigger than the Selecting a file event.

3.3 Textual description of Object to Object interaction

The GUI class serves as interaction with the user, as well as event flow control.

The fileReading class reads the file and extracts the information needed by GUI and Solver.

The Solver class pre - processes data, and implement the algorithm for solving the problem.

3.4 Subsystems

No subsystems required.

3.5 Implementation of Non – functional requirements

No non – functional requirements were identified.

3.6 Deployment model

The whole application is implemented on a single processor.

3.7 Legacy issues

No legacy issues as there are no major third party software components.

3.8 Reconsider the attributes

There was only a small change on attributes, which can be appreciated on the design class diagram in figure 9. A thread element attribute was added on the GUI class.

3.9 Reconsider the associations

No changes needed.

3.10 Statechart

No further revisions required.

3.11 Class diagram showing visibility

Design model class diagram with attributes and methods visibility

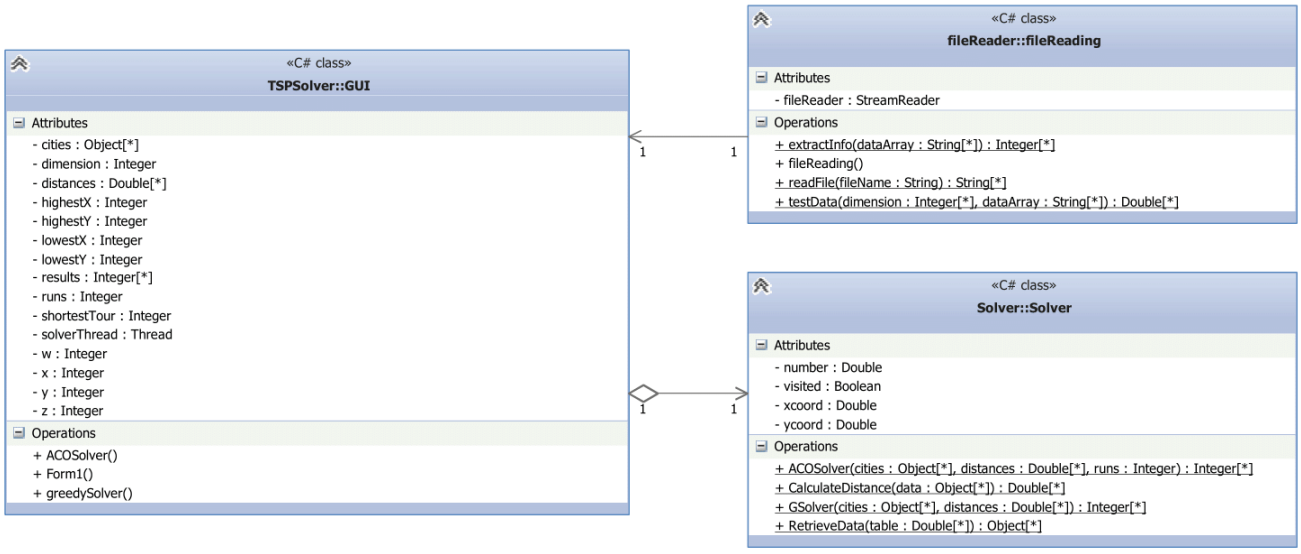


Figure 9 Design model Class Diagram showing visibility

Code implementation

This section describes each method of the designed classes, how it works and which type of data returns (if applicable). For convenience, this will be separated into classes.

Class: `public class fileReading`

Class fileReading contains three methods: readFile(), extractInfo() and testData().

Method: `public static string[] readFile(string fileName)`

Accepts a string fileName as a parameter, which contains the path of the selected file. It contains a StreamReader type attribute called fileReader. It uses an object of fileReading class in order to access the fileReader property of it.

Using an object of the FileStream class, it accesses the select file in read mode. It then stores each line of the file on a string array, where each index represents a line of text. Returns a string[] array with the data stored in it.

Method: `public static int[] extractInfo(string[] dataArray)`

Accepts a string[] dataArray as a parameter, which contains all the information of the read file. It loops in a while loop until the string "EOF" is found. Using a combination String class method Compare and method Split, it identifies the information needed for the GUI class which is filename, dimension and optimum tour length information. Returns a int[] array numericData with the number of cities and the number of the optimum tour length.

Method: `public static double[,] testData(int[] dimension, string[] dataArray)`

Accepts a string[] dataArray as a parameter, which contains all the information of the read file, and a int[] array dimension. The returned array tspData will have the same number of columns that dimension array. It loops in a while loop until the string "EOF" is found. Using a combination String class method Compare and method Split, and an enhancement by adding GetType method, it stores the coordinates of each city and its number on a double array. If the type of the first character in the dataArray is a integer number, then it stores the previous mentioned values. Returns a double[,] array tspData with each city coordinates and identifier.

Class: `public class Solver`

Class Solver contains four methods: CalculateDistance(), RetrieveData(), GSolver() and ACOSolver().

Method: `public static double[,] CalculateDistance(object[] data)`

Accepts an object[] array data. Each object represents a city. Using the objects attributes Xcoord, Ycoord, it constructs a table with the distances from each city with respect to the other cities. Returns a double[,] array with the distances between each of the cities with respect to others.

Method: `public static object[] RetrieveData(double[,] table)`

Accepts a double[,] array table. This array contains the coordinates and city number. Method create instances of Solver Class and store them into an object[] array, where each object represents a city. Returns an object[] array with the instances of Solver class.

Method: `public static int[] GSolver(object[] cities, double[,] distances)`

Accepts an object[] array cities and a double[,] array distances. Using this information, implements the Greedy algorithm, which based on the highest probability of the arcs, selects the nearest next unvisited city. It is noticeable that in this algorithm there is no pheromone update. The greedy algorithm can be idealized like an ACO of only one iteration (tmax =1). The algorithm simply sends an ant for each city, where Number of ants = Number of cities. Each ant starts at a different city, covering all of the cities at a starting point. The algorithm stores the best tour found and its path. Method GSolver returns an int[] array with the data of the shortest tour. Its length and the trajectory.

Method: `public static int[] ACOSolver(object[] cities, double[,] distances, int runs)`

Accepts an object[] array cities, a double[,] array distances and an int runs. Using this information, implements the ACO algorithm. This is the main algorithm of the project. Variable runs determines the number of iterations (tmax) which is controlled by a slider on the GUI. Hence the more the runs, better results but slower performance. Tuning runs can give good results with less iterations. The algorithm make local instances of the city for data handling. Each ant is assigned a random starting city. For practical reasons, the calculation of probability is divided within different arrays: For practical reasons, a small portion of the code will be attached to denote these arrays.

```
//Calculate the numerator factor for each arc, using Tij and Nij (both pheromone and
distance information)
                        for (int j = 0; j < gCities.GetLength(0); j++)
                        {
                            //Avoid division by zero
                            if (distances[currentcity, j] != 0)
                                nij = Math.Pow(1 / distances[currentcity, j], beta);
                            else
                                nij = 0.000001;
                            if (gCities[j].Visited == false)
                                decision[j] = Tij[currentcity, j] * nij;
                            else
                                decision[j] = 0; //Assign zero to a visited city
                            //Calculate local probability sum, which is the denominator
of the pij (sum of Tij * Nij)
                            localdecisionsum += decision[j];
                        } //End for j
```

Variable nij represents the influence of the arc length. That is nij = (1/dij)^beta

Array Tij[] represents the pheromone in each of the possible arcs.

Therefore, decision[] = Tij[]*nij. Where Tij represents the specific arc, and nij its length influence. Each entry decision represents the numerator for calculating the actual probability.

Variable localdecision sum represents the denominator of the probability calculation formula.

Array distances[currentCity, j] is used to prevent division by zero. This means that if the distance between City X and City X is compared, it should be 0 and a division by zero would take place. To prevent that, if that happens.

Next part, is to calculate probability, with:

```
//Calculate the real probability to go to each city
                        for (int k = 0; k < gCities.GetLength(0); k++)
                        {
                            if (gCities[k].Number == gCities[currentcity].Number ||
gCities[k].Visited == true)
                                pij[k] = 0;
                            if (gCities[k].Visited == false)
                                pij[k] = decision[k] / localdecisionsum;
                        } //End for k
```

Probability is calculated for the possible cities to visit, hence if the compared city is the current city, or the compared city has been already visited its probability goes to 0. Otherwise calculate probability.

To differentiate Greedy from ACO at this point, a cumulative probability array is calculated. This array helps to make the decision probabilistic, rather than the next nearest city. This procedure is explained with:

```
//Generate cumulative probability
                        for (int cumulative = 1; cumulative < acumpij.GetLength(0);
cumulative++)
                        {
                            acumpij[cumulative] += acumpij[cumulative - 1] +
pij[cumulative - 1];
                        }

//Generate random P number
                        highestProb = randomNum.NextDouble();
                        //Determine between which cities the probability is
                        while (highestProb > acumpij[s])
                        {
                                betweenLeft = s;
                                s++; //Increment cumulative probability array counter
                                if (s > pij.GetLength(0))
                                {
                                    betweenLeft = pij.GetLength(0) - 2;
                                    betweenRight = pij.GetLength(0) - 1;
                                    break;
                                }
                        }
```

Basically, a random probability is generated. Then, the while loop determines between which cities the probability is enclosed. This helps to get more choice of next city to visit, rather than only nearest P. When the two between cities are identified, the ant will choose then the nearest of the two cities. It is noticeable that the probability compared is random, and its value could be between a big number of different pairs of unvisited cities.

After that, the ant visit the city. Its memory is updated after each visit. After the ant complete the tour, its length its stored in a swarm array, which will be used later. A new ant is then repeating all of the process.

When a swarm end all of the tours, pheromone is triggered with

```
//Update pheromone values
                //Update Tij values
                //Part 1 sums all of the pheromone trails where each ant has been
                for (i = 0; i < antMemory.GetLength(0); i++)
                {
                    for (int j = 0; j < antMemory.GetLength(1) - 1; j++)
                    {
                        int index1 = antMemory[i, j] - 1;
                        int index2 = antMemory[i, j + 1] - 1;
                        deltaTij[index1, index2] += Q / paths[x - 1];
                    }

                } // End for Tij Part 1
```

```
                //Part 2 triggers evaporation rate
                for (i = 0; i < antMemory.GetLength(0); i++)
                {
                    for (int j = 0; j < antMemory.GetLength(1) - 1; j++)
                    {
                        int index1 = antMemory[i, j] - 1;
                        int index2 = antMemory[i, j + 1] - 1;
                        Tij[index1, index2] = ((1 - evaporationrate) * Tij[index1,
index2]) + deltaTij[index1,index2];
                    }

                } // End for Tij Part 2
```

Where the first part calculate the sums of all of the pheromone deposits from all the ants through a specific arc, and part 2 triggers the pheromone array update. Notice that this part uses the antMemory array, to deposit pheromone on the actual trail arcs.

With logic control, the algorithm stores the shortest path and tour of each swarm. Finally, the algorithm selects the shortest tour and length of all of the iterations, giving the best result found on the run.

Class: `class GUI`

Class GUI contain the event handlers and the processes for the graphics update, as well as the file opening and solver selection menus. This class will be described with another approach. A screenshot of the main interface will help to describe functionality.

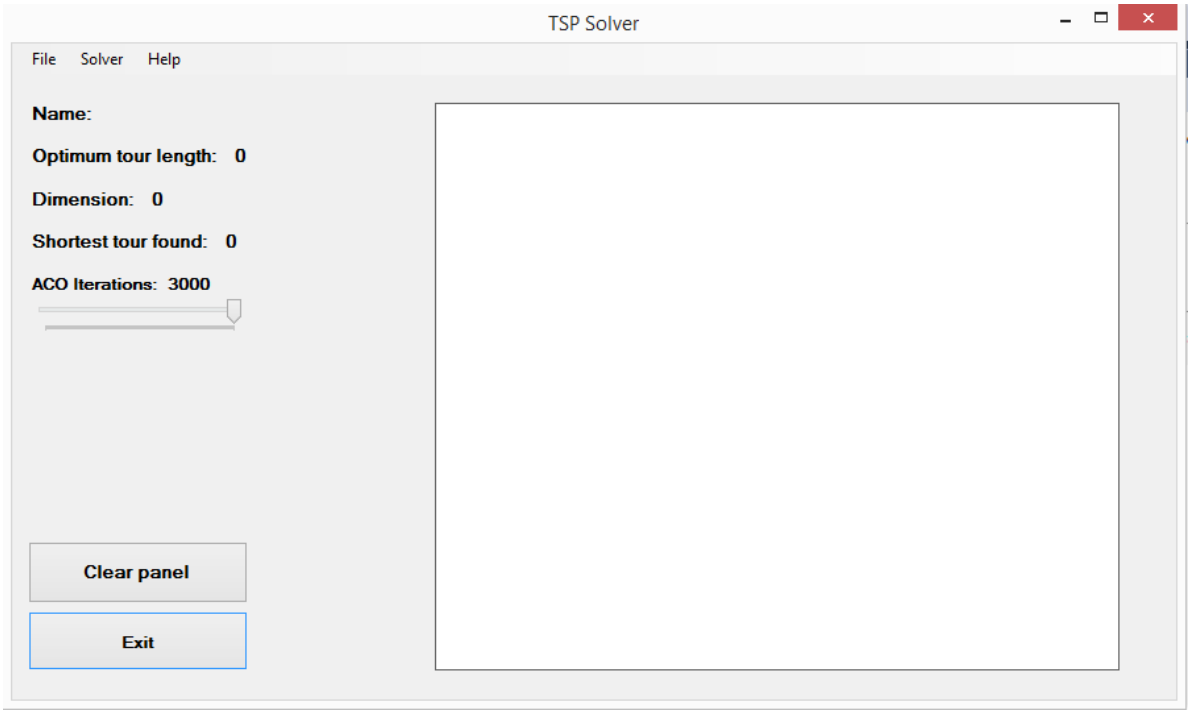


Figure 10 Application Main Window

The Main window contains a menu which lets the user:

- Select and open a file
- Select a solver
- Read about
- Exit the application

It contains two buttons:

- Clear panel
- Exit

It contains also a slider.

Describing the elements:

When a file is opened, the screen is updated, and the initial information is displayed. By using methods from the fileReading and Solver Class, information is obtained and processed. When a user selects a file, first the fileReading class extracts the information and the relevant labels are updated. Then, the Solver Class pre processes the information. With this information, GUI class makes use of the Graphics class, to feed a Point type array for map graphing. It first stores the points. Then it determines the lowest and highest X and Y values. The reason is because in order for every map to fit, a re-scaling is needed. When values are rescaled, then with the use of an object of the Graphics class and the method FillRectangle, points are drawn at the main panel. Figure 11 shows a graphical correlation with previous description.

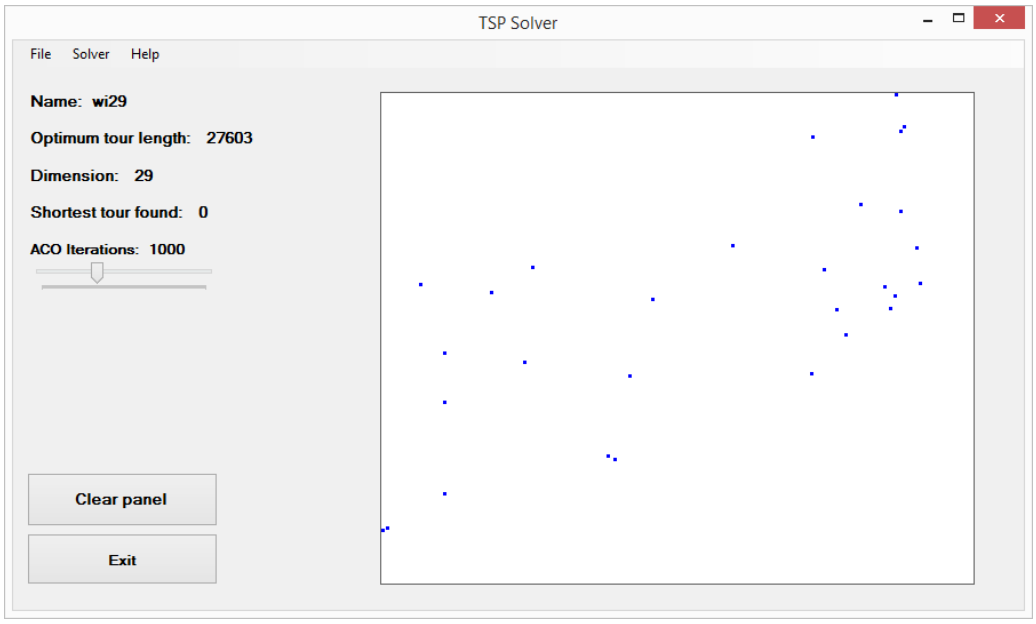


Figure 11 Main GUI after file opening

When selecting a solver, the application targets the problem with the selected approach. Further screenshots in the Testing section will demonstrate that. After completing the algorithm, the path is drawn on the map using the Graphics library and method Draw Line. The path is order is stored on an array called results that store the trajectory and the dimension. After completion, GUI uses the result of the solver for updating the screen.

There are some key points to notice. When an algorithm is running, a new thread is initiated, so that while the application computes the results, the user can still interact with the application. When a user presses the exit button at current algorithm, the application terminates. If a user opens a new file while algorithm is running, the current thread stops and a new dataset is prepared.

Multi-threading generated a problem: A label that was created in a certain thread could not be accessed when another thread was running. This was fixed by using the Invoke method that executes a call back on the GUI thread that has the correct text.

The slider again, controls the number of iterations of the ACO algorithm. The bigger the number is more likely to get better results but sacrificing speed. So the user can control to which aspect give more weight.

The rest of the functionality is driven by event handling, which makes use of automatically generated code as well.

Testing

Some testing procedures were executed to prove functionality of the application. The same datasets where tested with the different algorithms, and the results were compared. It was noticed that for big data sets, the ACO algorithm slowed down significantly. The next images show a few data sets compared with both algorithms, and ACO varying with 1000 and 2000 iterations.

The test shown was made with:

- Wi29 dataset
- Dj38 dataset
- A custom dataset of 5 cities (name TestSet1, included on TSP Data folder)

Wi29 dataset (images next pages):

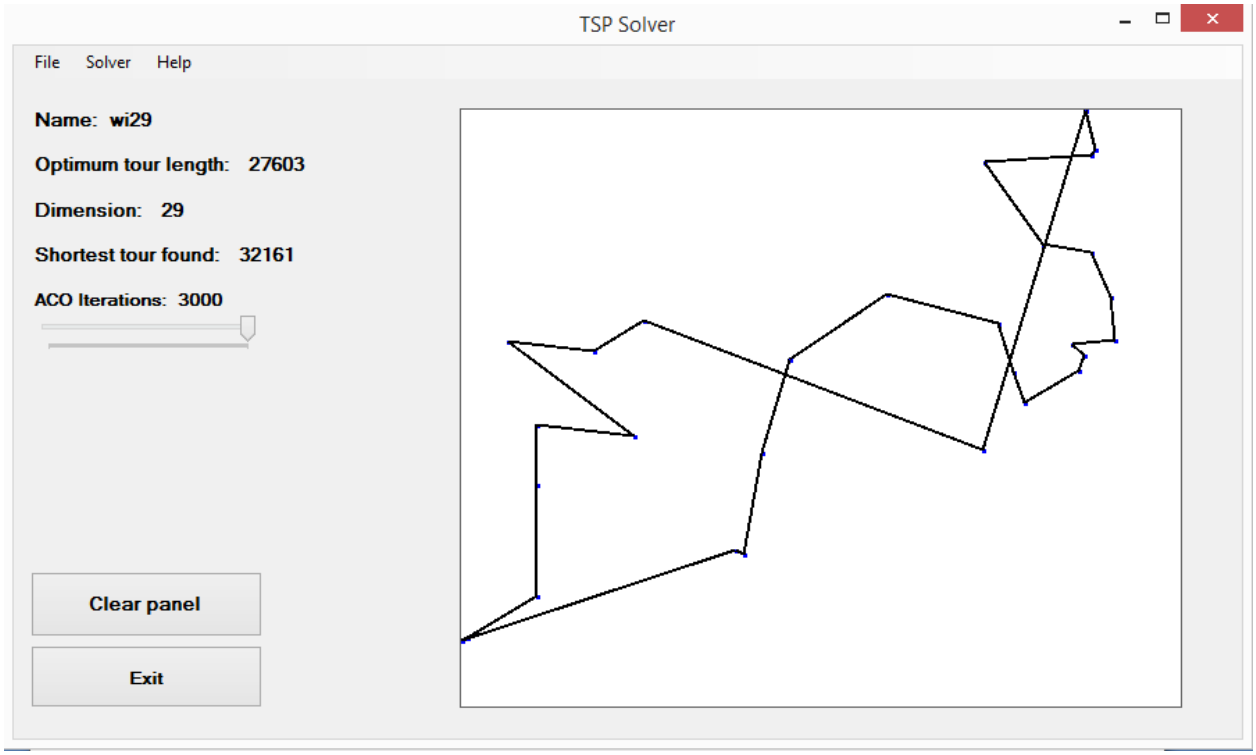


Figure 12 wi29 tsp Greedy Algorithm

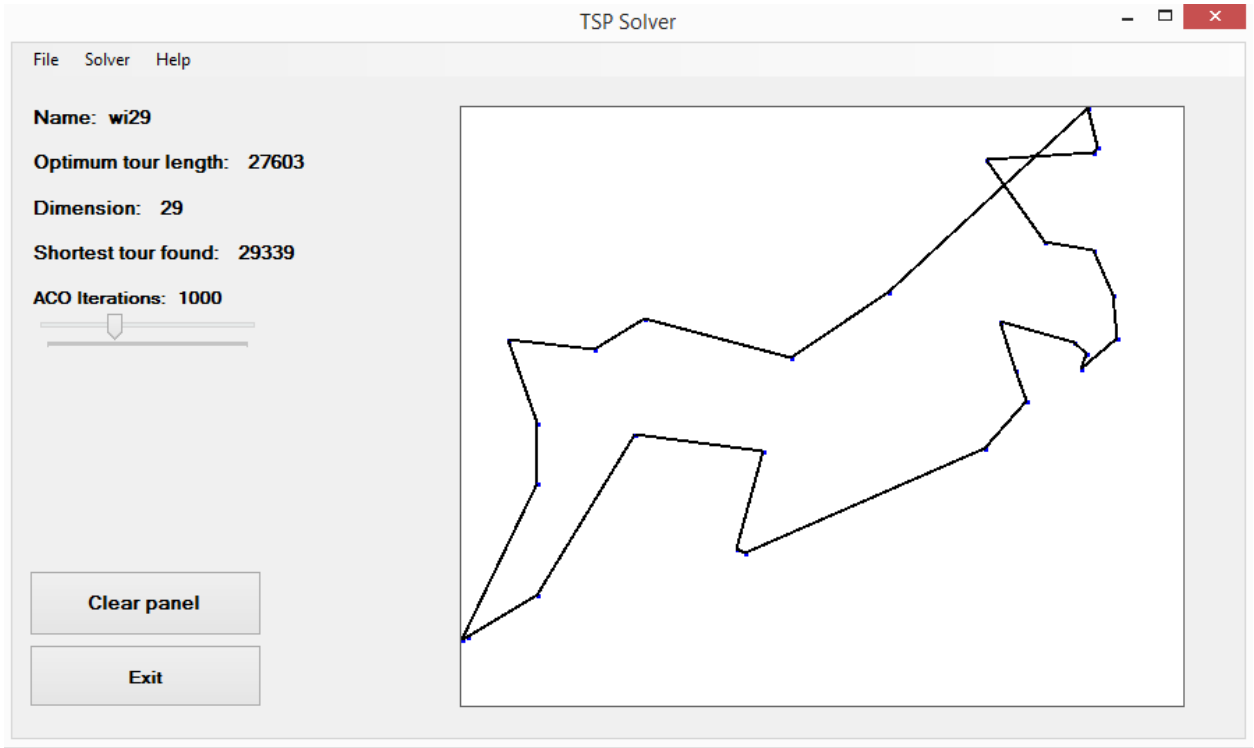


Figure 13 wi29 dataset ACO 1000 iterations

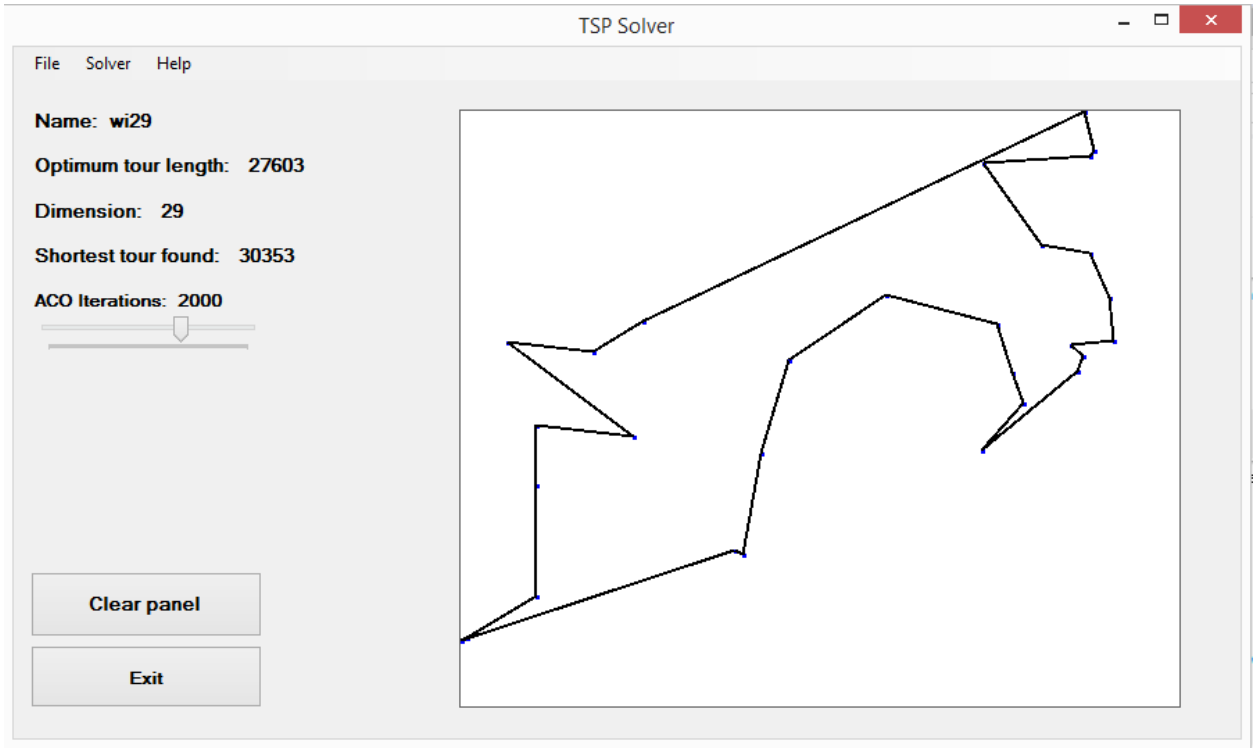


Figure 14 wi29 dataset ACO 2000 iterations

Dj38 dataset (images begin next page):

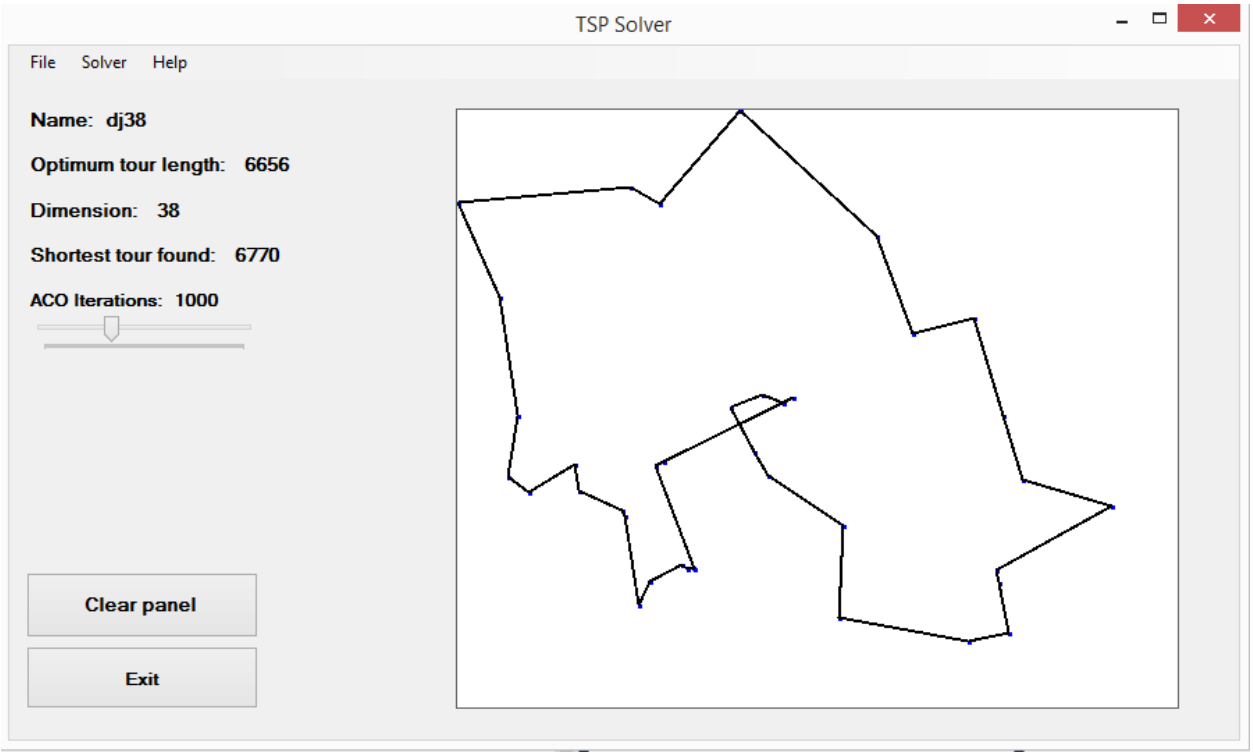


Figure 15 dj38 testset Greedy Algorithm

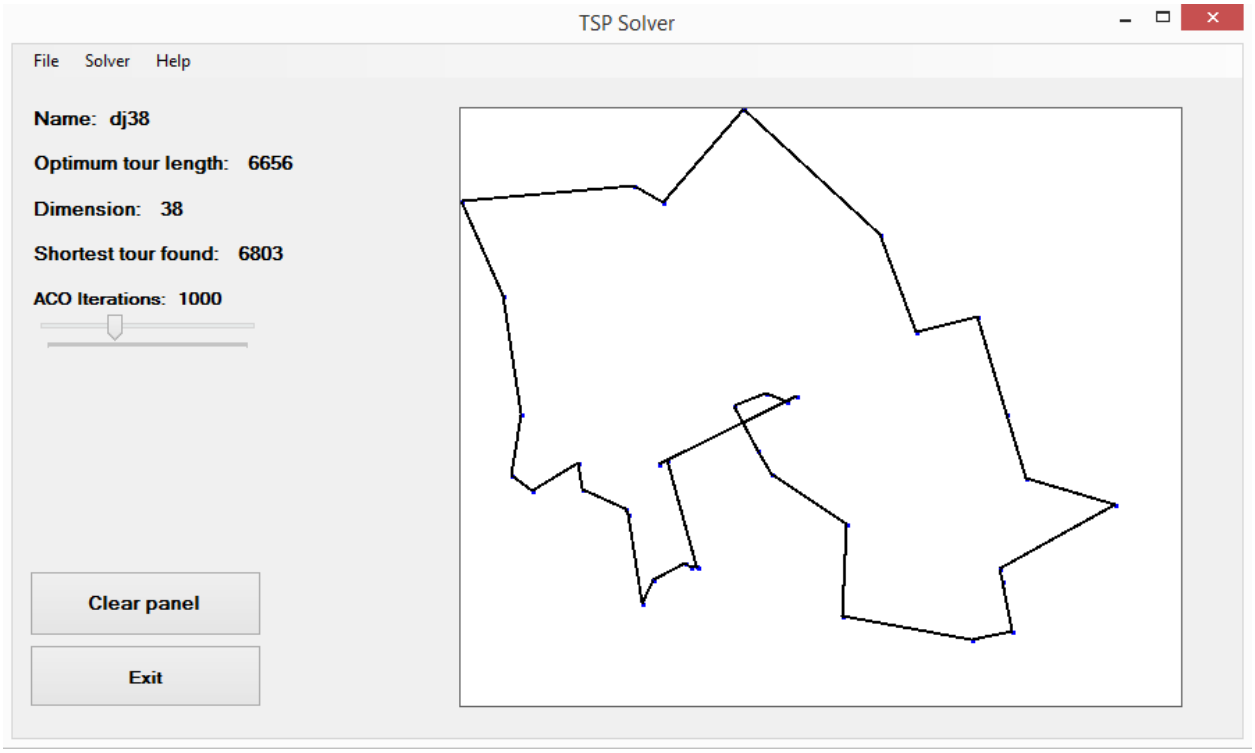


Figure 16 dj38 testset ACO 1000 iterations

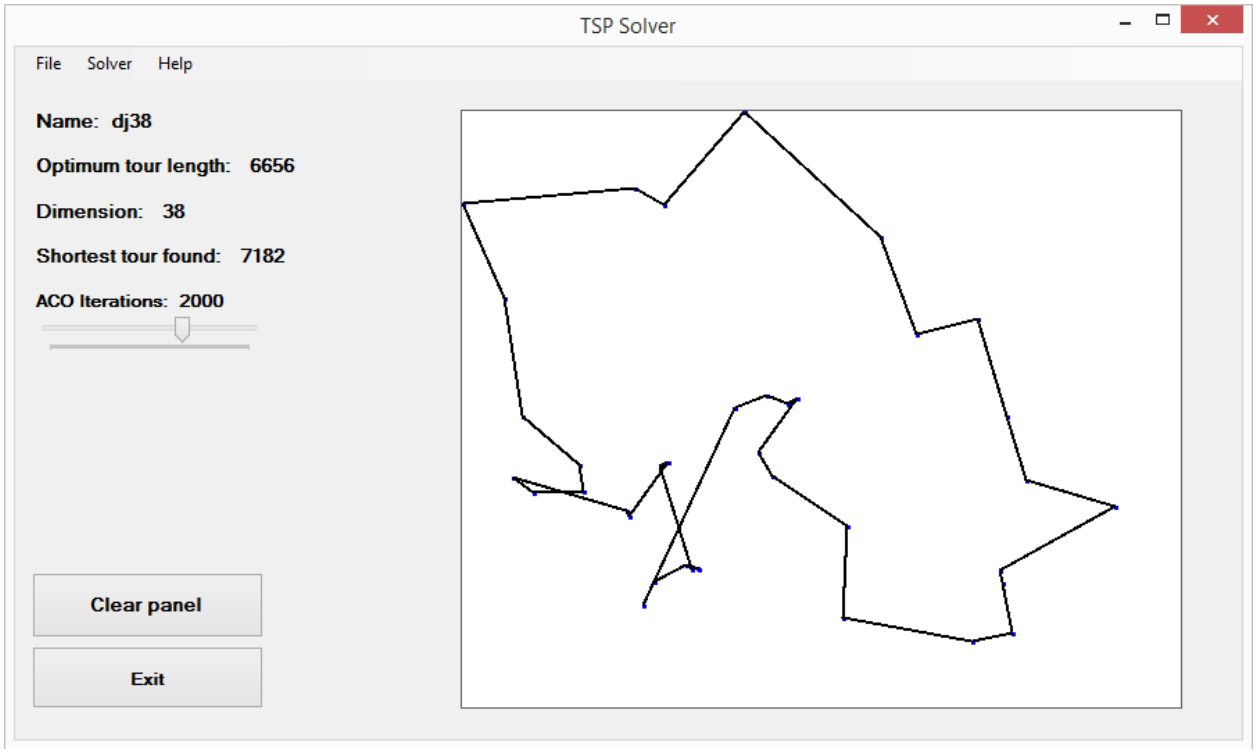


Figure 17 dj38 testset ACO 2000 iterations

Custom Test set:

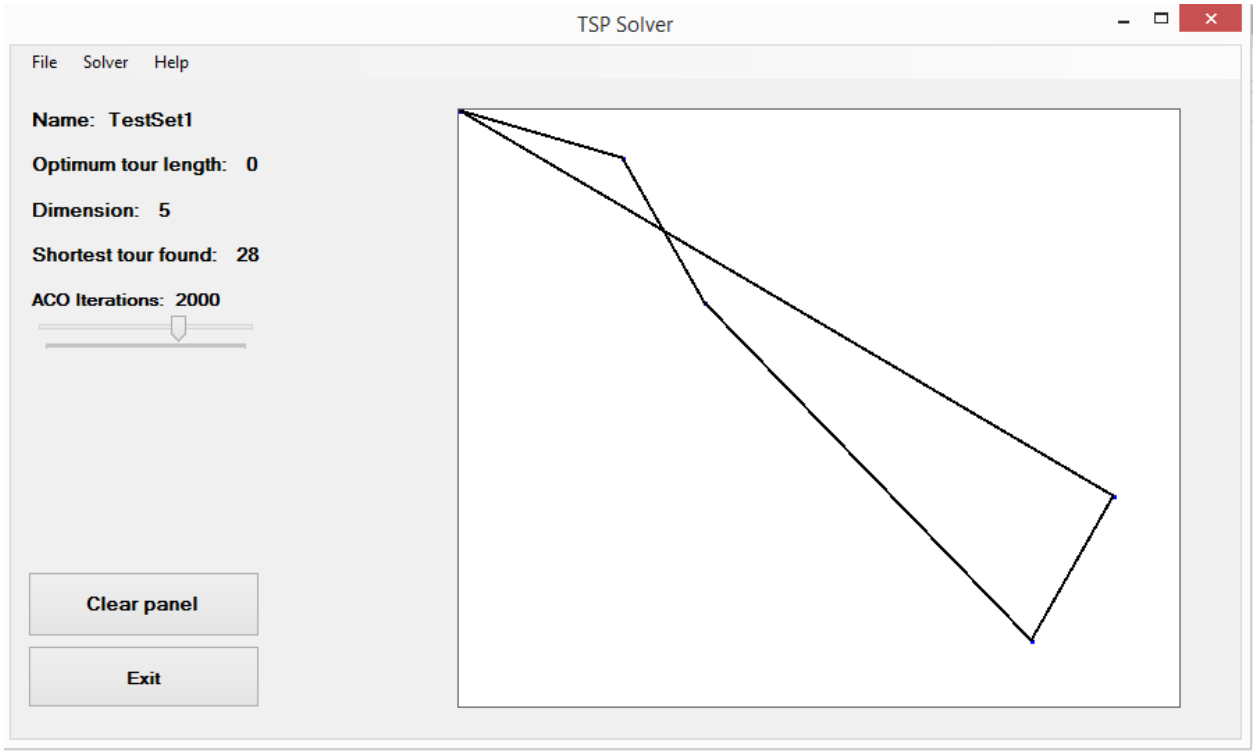


Figure 18 Custom testset Greedy

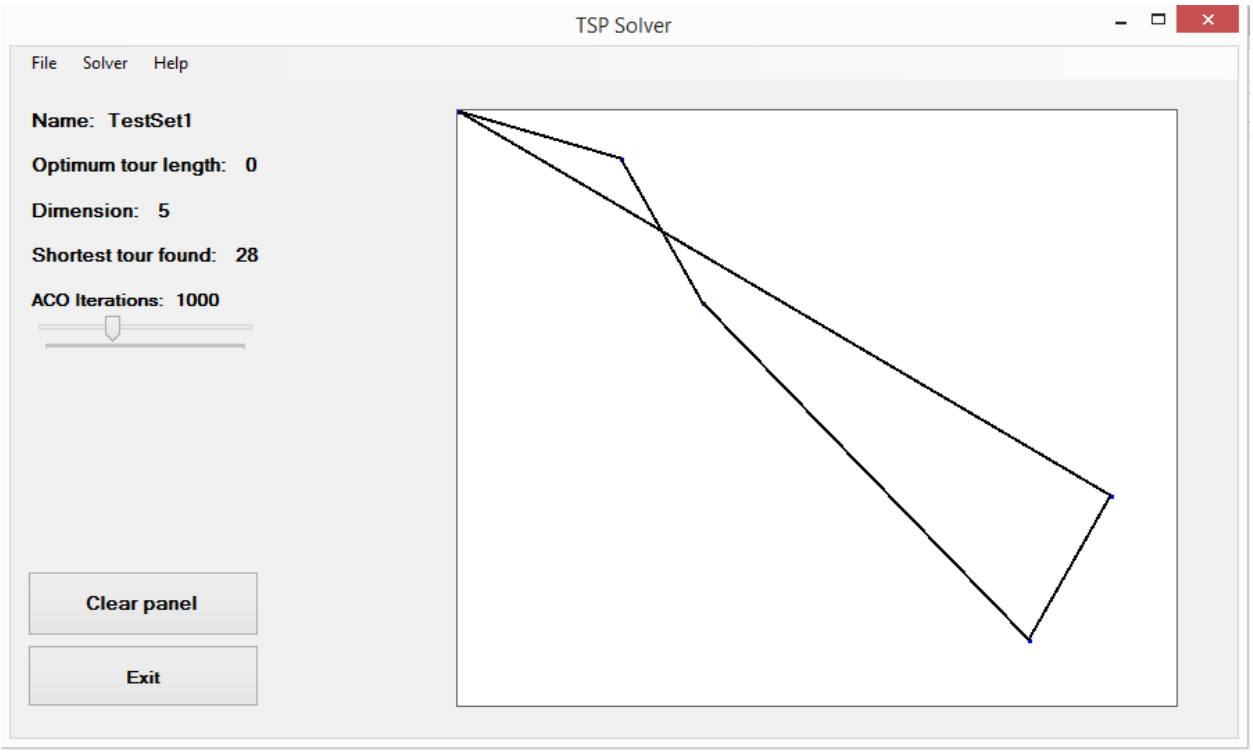


Figure 19 Custom testset ACO 1000 iterations

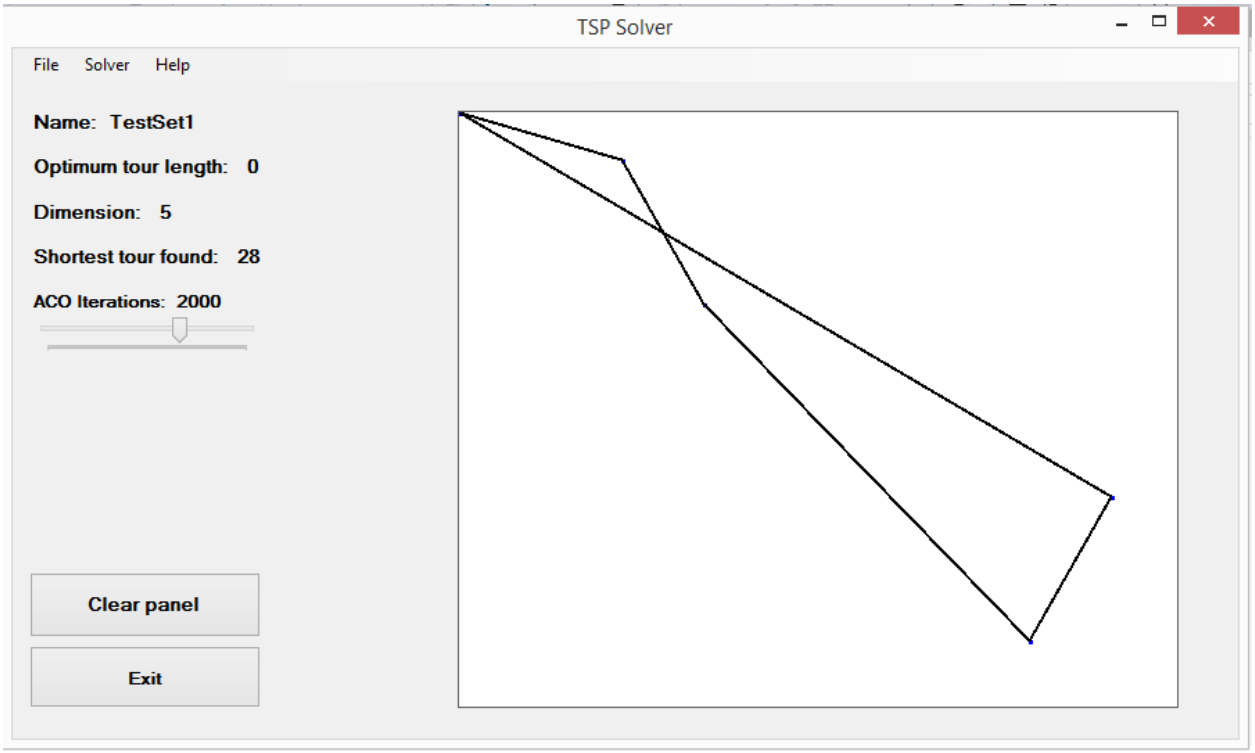


Figure 20 Custom testset ACO 2000 iterations

Conclusions

Ant Colony Optimization is a powerful algorithm that can be potentially refined via the application of additional methods. The current design applied only the basic version of the ACO. The greedy and ACO algorithm tend to get similar results when ACO iterations are really low or the number of cities is really small (that could be observed on the custom test set). These two factors play a vital role on the efficiency of the designed program. For cities N<50, the greedy is incredibly fast, and depending on the number of iterations, the ACO is fairly fast as well. The difference in results also may vary depending on the city distribution. With the 29 city dataset, the ACO had significantly better results with 1000 and 2000 iterations than the greedy, being 1000 the better one. For the case of the 38 city set, greedy had a better performance than the ACO, with 1000 and 2000 iterations as well. It is noticeable that ACO had better results with 1000 iterations.

The notorious difference in speed could be observed when the 194 city set (not shown) was tested. The greedy could get a result in approx 12 seconds (experimental time), but the ACO was really slowed down, even for iterations in the range of 40 - 50. Improvements to the design can include implementing the daemon process to the ACO, or giving the user the option to tune the parameters of the ACO, in order to have a better control on the efficiency of the program.

In overall performance, the program met quite well the specification, as it reports good results in considerably fast times. This means that a big achievement was done, as the ACO was effectively implemented to attack the TSP.

This task was achieved by applying new learned concepts, such as design procedures in software engineering, object oriented programming concepts (managing objects, classes, properties, methods), graphical user interface programming (event handling), and use of the programming language C#.

References

Deitel, P., & Deitel, H. (2009). *Visual C# 2012 How to program.* Pearson.

Dorigo, M., & Stützle, T. (2004). *Ant Colony Optimization.* Massachusetts Institute of Technology.

Teodorovic, D. (2008). Swarm intelligence systems for transportation engineering: Principles and applications. *Elsevier*, 651–667.

Figures

Figure 1 Use-Case Diagram 3
Figure 2 Class Diagram (Stereotypes) 5
Figure 3 Statechart Diagram 6
Figure 4 Collaboration Diagram 7
Figure 5 Sequence diagram 9
Figure 6 Revised Sequence diagram 13
Figure 7 Analysis Class Diagram 14
Figure 8 Design model Sequence Diagram 16
Figure 9 Design model Class Diagram showing visibility 17
Figure 10 Application Main Window 22
Figure 11 Main GUI after file opening 23
Figure 12 wi29 tsp Greedy Algorithm 25
Figure 13 wi29 dataset ACO 1000 iterations 25
Figure 14 wi29 dataset ACO 2000 iterations 26
Figure 15 dj38 testset Greedy Algorithm 27
Figure 16 dj38 testset ACO 1000 iterations 27
Figure 17 dj38 testset ACO 2000 iterations 28
Figure 18 Custom testset Greedy 28
Figure 19 Custom testset ACO 1000 iterations 29
Figure 20 Custom testset ACO 2000 iterations 29

Tables

Table 1 Class Responsibility Collaboration Cards 5
Table 2 Attributes 10
Table 3 Methods 11