

**The University of Birmingham**  
**Department of Electronic, Electrical and Systems**  
**Engineering**



**Object Oriented Programming Using C#**  
**Swarm Intelligence**

Luis Enrique Ramos Maldonado

1487473

December 2014

Professors

Dr Mike Spann

Mr David Pycock

## Contents

|   |    |
|---|----|
| Table of Figures .....  | 2  |
| Tables.....   | 2  |
| I. Introduction .....   | 3  |
| II. Use-Case Model.....   | 4  |
| II.1. Requirements Analysis .....                                     | 4  |
| II.2. Use-Case Diagram .....  | 5  |
| II.3. Scenario Description .....                                      | 5  |
| II.4. Class-Responsibility-Collaboration Cards and Class Diagram..... | 6  |
| II.5. Statechart Diagrams.....  | 8  |
| II.6 Interaction Diagrams.....  | 8  |
| III. Analysis Model.....  | 11 |
| IV. Design Model.....   | 14 |
| IV.1. Code Flow.....  | 16 |
| V. Testing.....   | 17 |
| VI. Conclusions.....  | 20 |
| Bibliography .....  | 21 |

## Table of Figures

|  |    |
|--|----|
| Figure 1. Ant colony optimization example. Source: (Bachir, Ali, & Abdellah, 2012) ..... | 3  |
| Figure 2. Use-Case Diagram .....   | 5  |
| Figure 3. Use Case Model Class Diagram .....   | 7  |
| Figure 4. Use-Case Model Statechart Diagram .....  | 8  |
| Figure 5. Collaboration Diagram .....  | 9  |
| Figure 6. Sequence Diagram .....   | 10 |
| Figure 7. Analysis Model Statechart Diagram .....  | 12 |
| Figure 8. Analysis Model Sequence Diagram .....  | 13 |
| Figure 9. Analysis Model Class Diagram .....   | 14 |
| Figure 10. Design Model Sequence Diagram .....   | 15 |
| Figure 11. Design Model Class Diagram .....  | 16 |
| Figure 12. Test Set .....  | 17 |
| Figure 13. 29 Cities Test .....  | 18 |
| Figure 14. 194 Cities, before running solver .....                                       | 18 |
| Figure 15. 194 cities, after solver .....  | 19 |
| Figure 16. 929 cities result .....   | 19 |

## Tables

|                          |   |
|--------------------------|---|
| Table 1. CRC Cards ..... | 7 |
|--------------------------|---|

## I. Introduction

This programming assignment is centered on swarm intelligence, which is a type of artificial intelligence which implements intelligent systems based on the behavior of insects such as ants, termites, and bees (Passino & Liu, 2000). The focus was on imitating ant colonies and how they communicate between themselves when they need to get to a food source. It is well known that ants communicate via a pheromone deposit in order to indicate to other ants where they should go in order to travel from the nest to the food source (Panait & Luke). Ant colony optimization (ACO) algorithms have been studied for many years as they pose an interesting and feasible solution to discrete optimization problems. Figure 1 presents us with an explanation of how ACO works, when a group of ants identify a food source, they all go after it using whatever path each individual ant deems best, however after some time there will inevitably be paths that are more frequented than others, thus having a higher amount of pheromones, which in turn attracts more ants towards it, eventually leading all ants through the most efficient path.

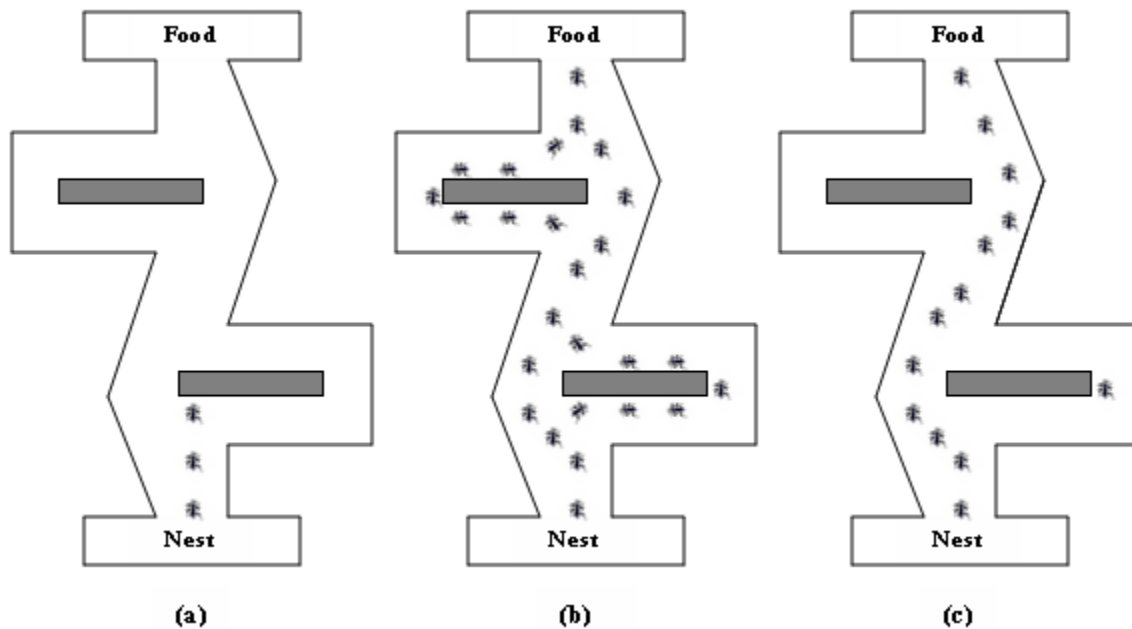


Figure 1. Ant colony optimization example. Source: (Bachir, Ali, & Abdellah, 2012)

In this assignment, however, the goal wasn't to use the ACO algorithm to find the best path to a food source; instead, the goal is to implement it as a means to solve the Travelling Salesman Problem (TSP).

The travelling salesman problem is a classic optimization problem where we are given a series of cities and we have to find the most efficient route to travel to every city without repeating a city (except for the starting city, which is where the tour ends) (Applegate, Bixby, Chvátal, & Cook, 2007). As we can see, it's easy to see how the ACO can be useful in solving this problem. There are two main ways in which the ACO can be implemented; the first is using a "Greedy" algorithm, often referred to as the "Nearest Neighbor Algorithm", this algorithm works by making as many tours as there are cities, each tour starting in a different city, the way it decides which city to go to next is decided simply by distance, it always goes to the nearest city not yet visited, which is easy to implement, but not as efficient as other solutions (Liu, Moore, Gray, & Yang). A more refined version of the ACO, however, uses probability to determine what city to go to next; this probability is affected by the distance of the city from the current position, the number of cities around the current position that are unvisited and the influence of pheromones on the trails. Every iteration of the ACO a number of ants equal to the number of cities are released and base their tour off of the above probability, at the end of the iteration the ants release their pheromone deposits, which are determined by the length of the tour that has been undertaken, and the pheromone of the ants on the previous iteration begins to evaporate, this is to prevent all ants from ending up on the same tour, which would cause stagnation. Thus, the ACO algorithm results in a better path the more iteration that are made, but as the number of iterations increase, so does the time required to run it, so the user must decide what he needs, a fast result or a more efficient one (Dorigo & Stutzle, 2004).

## II. Use-Case Model

### II.1. Requirements Analysis

Since the assignment was geared for object oriented programming, some object oriented design had to take place. The first step in the design was a quick brainstorming session to determine the potential requirements, these are just to give an idea of how to start designing the program, so they serve as just an outline, but still help identify some aspects we would not have thought of otherwise. The potential requirements analysis that was made for this use case model was the following:

- Program must read the data in the files provided and display it to the user in an easy-to-read way.
- There must be different ways to solve the TSP.
- Graph the city coordinates after loading a map file.
- Draw the most efficient tour found after running an algorithm.
- Use multi-threading to avoid hang-ups and freezing screens.
- Pre-processing should occur right after loading files to speed up solving algorithms.
- When running the ACO algorithm, ants should leave a pheromone behind their trail to indicate they've been there.
- If a new map is loaded while a solving algorithm is running on another one, the program must suspend the one currently running and load the map that was just selected.

Since there is no great risk associated with any of the additional requirements, the design will then incorporate all of them.

## II.2. Use-Case Diagram

Once the potential requirements are analyzed, the next step was taken, which was to create the initial Use-Case Diagram where the relationship between the actor and the program are defined. In this case there was just one actor identified which is the User controlling the program, the Use-Case Diagram can be seen on Figure 2.

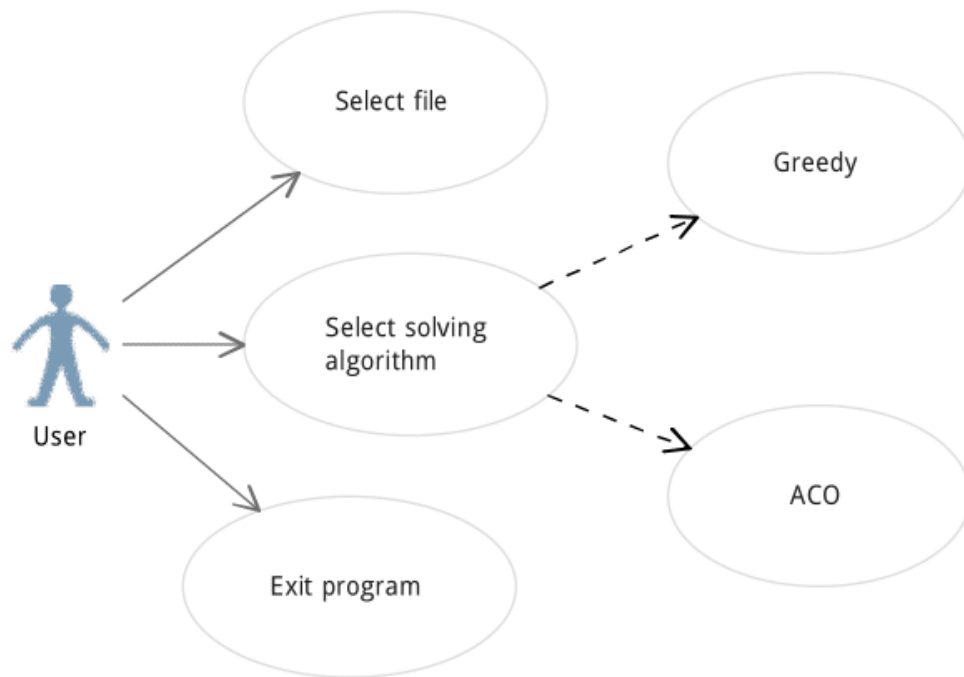


Figure 2. Use-Case Diagram

As we can see, the user can really only interact with the program by doing one of three things, either selecting the file to be solved, choosing which algorithm to use to solve the TSP (which in this case can be either the Greedy algorithm or the ACO algorithm), or exiting the program. Even though this is a very basic diagram it provides a visual representation of the limitations of the user, which in turn helps determine what tasks must be taken care of by the program, which then helps determine what must be programmed. In order to help take this further, scenario descriptions were then made.

## II.3. Scenario Description

In the scenario description, the aim is to try to explain what happens in the system when the user interacts with it in various ways, in order to expand on the Use-Case Diagram and help detect possible

aspects that had been overlooked and identify the classes that will be used in our program. The scenario description is the following:

**User:** The user will select a map file from an open file menu, after opening the file, its contents will be read and processed in order to optimize the information for use with the solving algorithms, at the same time, relevant contents such as the file's name, number of cities, optimum tour length, and a graph showing the cities in the map will be displayed to the user. When this process is finished, the user will then select one of the algorithms to solve for the TSP, after doing so, the program will execute the selected algorithm using the pre-processed data, when it's done, the result, the shortest path found, will be displayed to the user both as a numerical value as well as graphically. When this process is done, the user will select a new algorithm to solve for the TSP, after which the same process is followed. If the user selects a new map while already running an algorithm, the program will stop the algorithm currently running and begin the process anew from the new file selected. The program ends if the user clicks on the 'Exit' button.

As we can see, various things have now been taken into consideration that had not been before, such as the pre-processing of data and the graphical output of results. From this, it was determined that at least three classes would be needed to fulfill the requirements, which were named: DataFiles, Solver and GUI.

## **II.4. Class-Responsibility-Collaboration Cards and Class Diagram**

CRC Cards help identify the relationship between the classes that were just defined, attributes, and methods; therefore they are a very useful tool in the design of object oriented programming. At this point, as we can see in Table 1, they are still basically outlines of their basic functions and relationships:

Table 1. CRC Cards

|   |                      |
|---|----------------------|
| <b>Class: DataFile</b>  |                      |
| <b>Responsibilities</b>   | <b>Collaborators</b> |
| The DataFile class is responsible for reading the data off of the user selected file. It grabs the file name, number of cities in the map, optimum tour length and the cities' coordinates. After getting the coordinstes it sorts all the cities by distance in order to help with the processing time of the solving algorithms.  | GUI                  |
| <b>Class: Solver</b>  |                      |
| <b>Responsibilities</b>   | <b>Collaborators</b> |
| The Solver class is responsible for getting the pre-processed data from the DataFile class in order to solve for the TSP using the user-selected algorithm. After it's done with the algorithm, it passess the results over to the GUI in order to display them to the user.  | GUI, DataFile        |
| <b>Class: GUI</b>   |                      |
| <b>Responsibilities</b>   | <b>Collaborators</b> |
| The GUI class is responsible for being the medium between the user and the rest of the program. It will show the information extracted from the DataFile class as well as plot the cities using their coordinates. When the Solver class is done running an algorithm, it will update the information shown to the user, with the shortest path found as well as the tour itself on the same graph as the cities. | Solver, DataFile     |

The class diagram helps us structure our program better by providing a visual aid as to what composes our classes, for the Use-Case Model the Class Diagram is shown in Figure3.

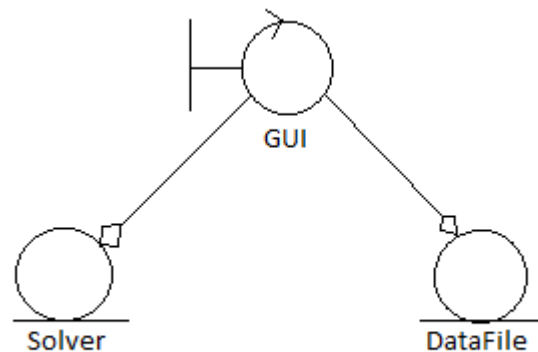


Figure 3. Use Case Model Class Diagram



## II.5. Statechart Diagrams

The statechart is used to identify the different states that our system will be in and the transitions necessary to go from one state to another. This helps further determine what actions can be done at each stage, thus improving the design. Figure 4 shows the statechart diagram for the current design.

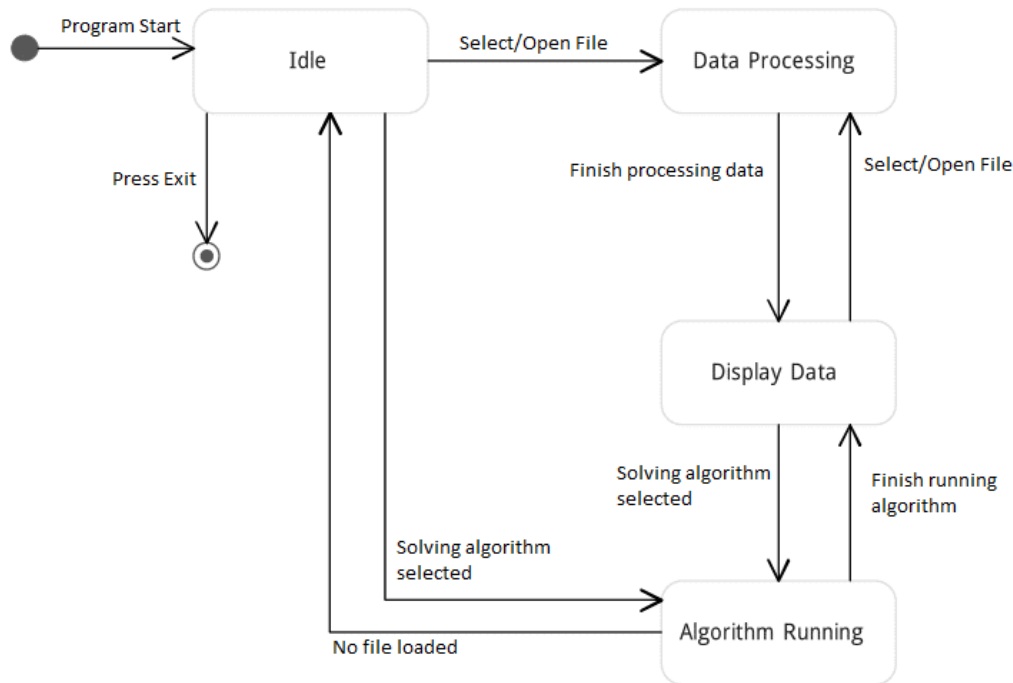


Figure 4. Use-Case Model Statechart Diagram

As we can see, there are four states for the system: idle, data processing, display data, and algorithm running, this helps visualize the system running and makes a useful tool for determining what actions, or processes can be done at each state, rather than think about all the processes that should be made and risk missing one or more processes.

## II.6 Interaction Diagrams

The final steps in the Use-Case Modeling phase of the design process are to draw the interaction diagrams, which help determine message interaction between classes. The first diagram presented, Figure 5, is a collaboration diagram, which outlines the direct interactions between classes, allowing room to start thinking about the sequence in which messages should be sent and received.

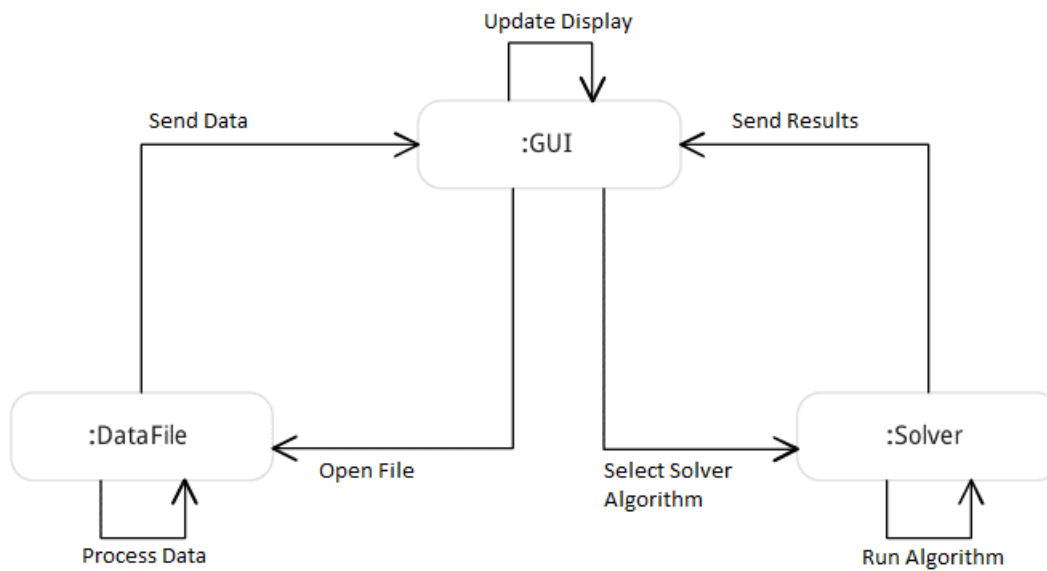


Figure 5.Collaboration Diagram

Once the collaboration diagram is done with, as was said before, the sequencing of the messages and events have to be considered, which are outlined in the appropriately named Sequence Diagram in Figure 6, which expands on the collaboration diagram by showing what messages and actions are sent or made and when .

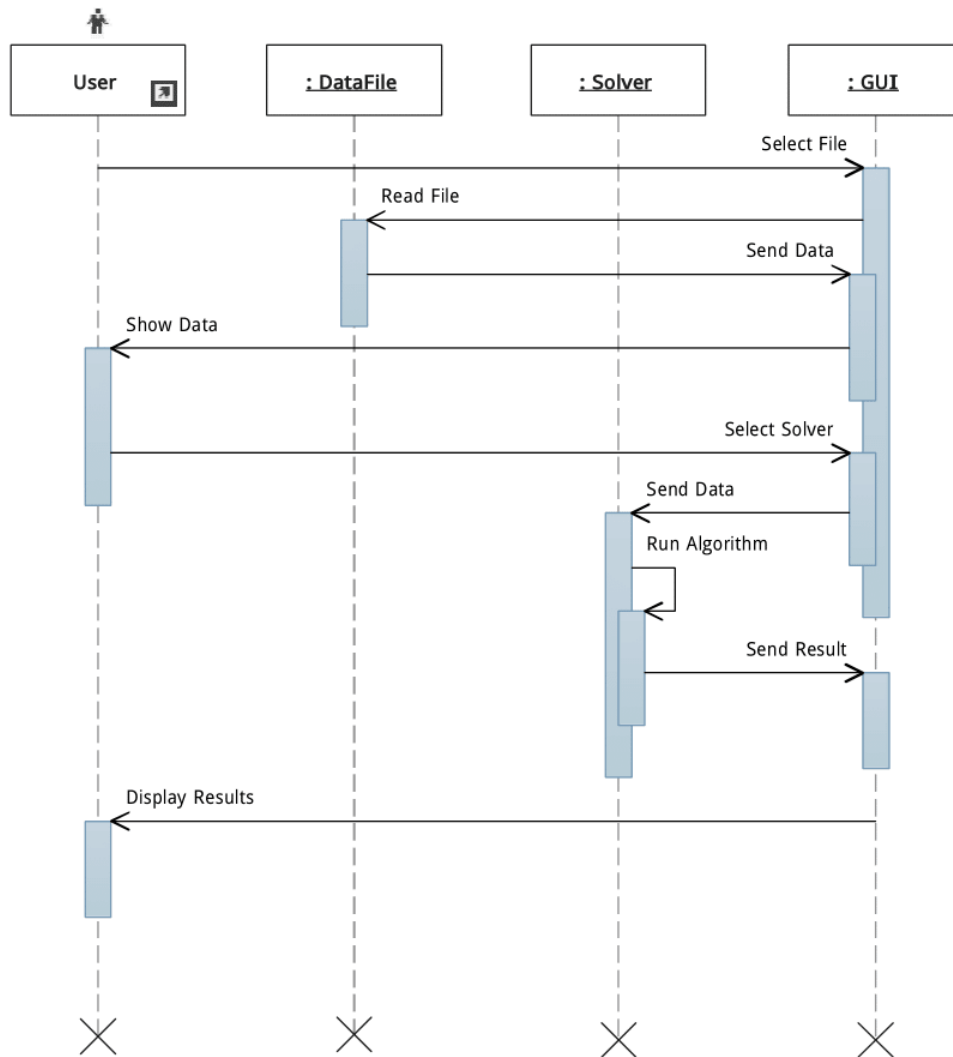


Figure 6. Sequence Diagram

### III. Analysis Model

The Use-Case Model helps identify the functionality that is required; the analysis model is concerned with identifying how that functionality can be achieved. Based on the whole Use-Case Model and some general reflection, each class's methods and attributes can be determined:

#### DataFile Class

Potential attributes: Dimension, Name, Tour

Potential methods: ReadFile( ), PreProcessing( ), Sort( )

#### Solver Class

Potential attributes: ShortestPath,

Potential methods: Greedy( ), ACO( )

#### GUI Class

Potential attributes:

Potential methods: OpenFile( ), SelectSolver( )

Based on this information, the diagrams that have been used up to this point can be re-drawn in order to be more specific and including these attributes and methods.

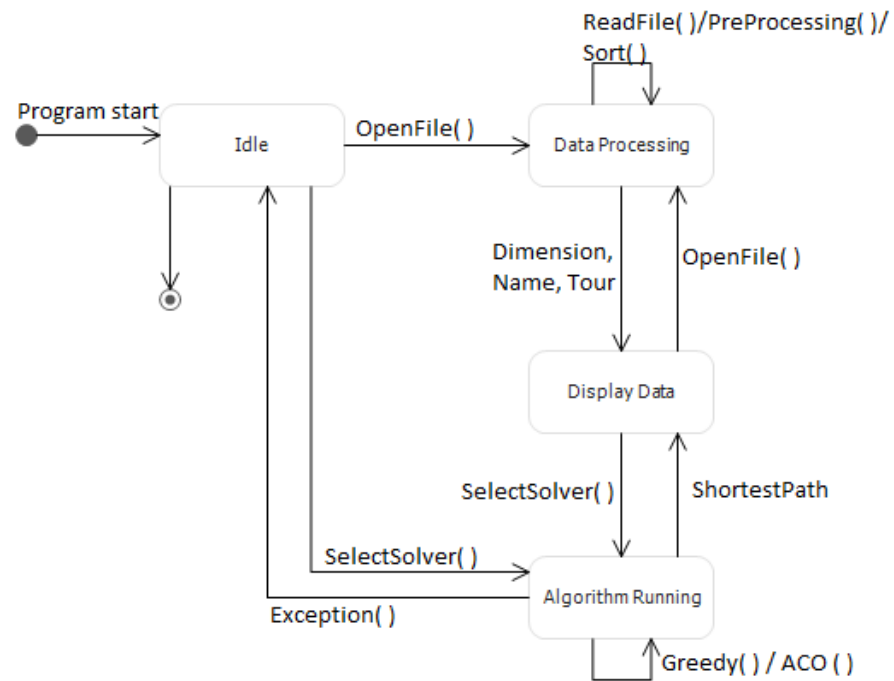


Figure 7. Analysis Model Statechart Diagram

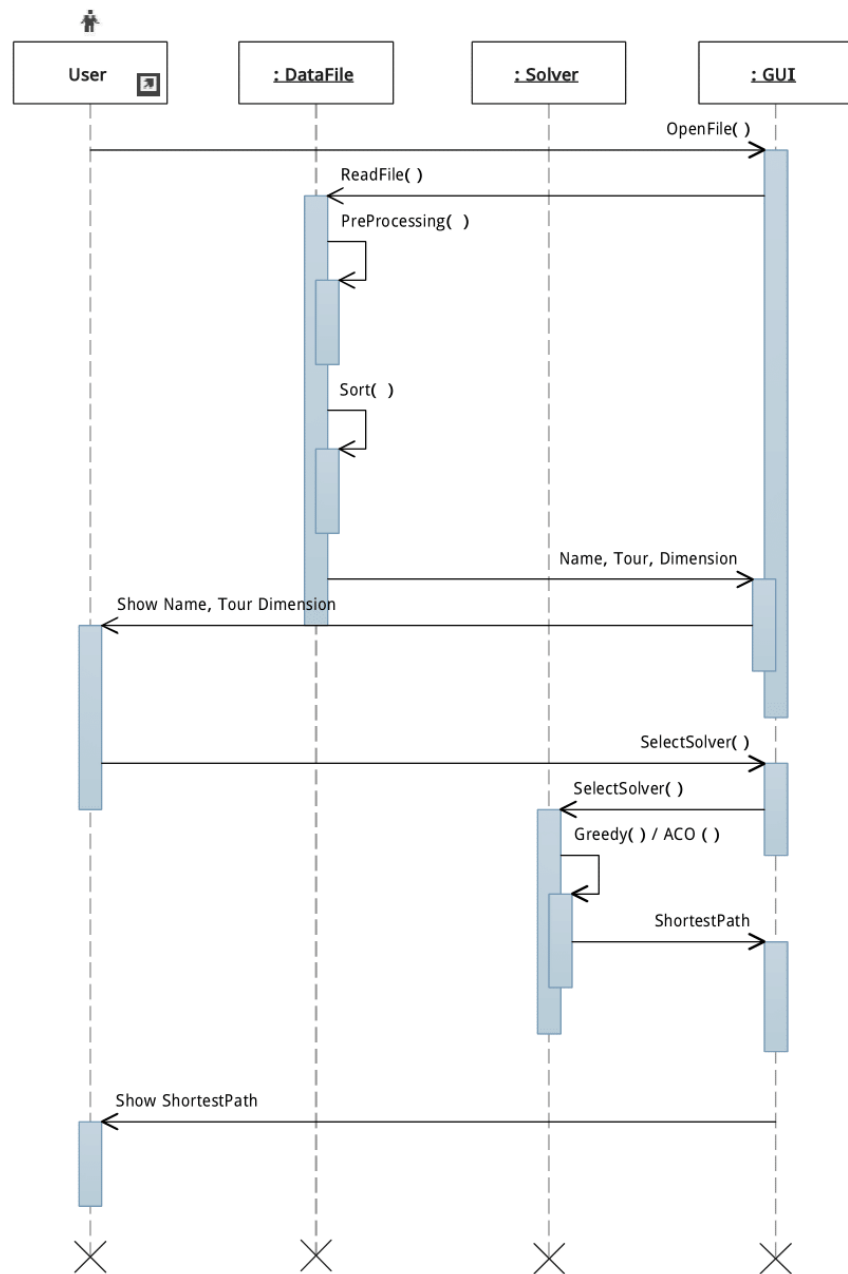


Figure 8. Analysis Model Sequence Diagram

Based on the previous diagrams, we can determine the type of each attribute we've defined, which will be added on to the class diagram, as well as the methods that each class contains, as illustrated in Figure 9.

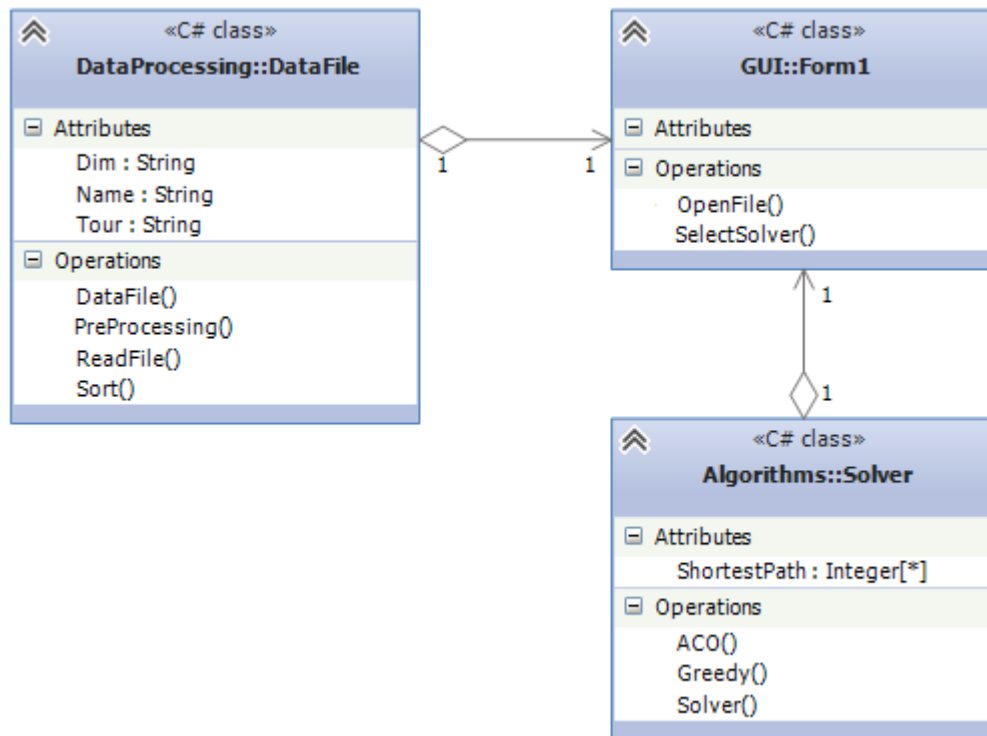


Figure 9. Analysis Model Class Diagram

#### IV. Design Model

After going through the Analysis Model, there were a few methods that were detected that were missing from some of the classes, namely, the GUI class needed multithreading to be able to operate at the same time as an algorithm was running, and the Solver class needed a reference table method to process some data while solving the algorithms. With this in mind, the final revision of the diagrams could be completed.

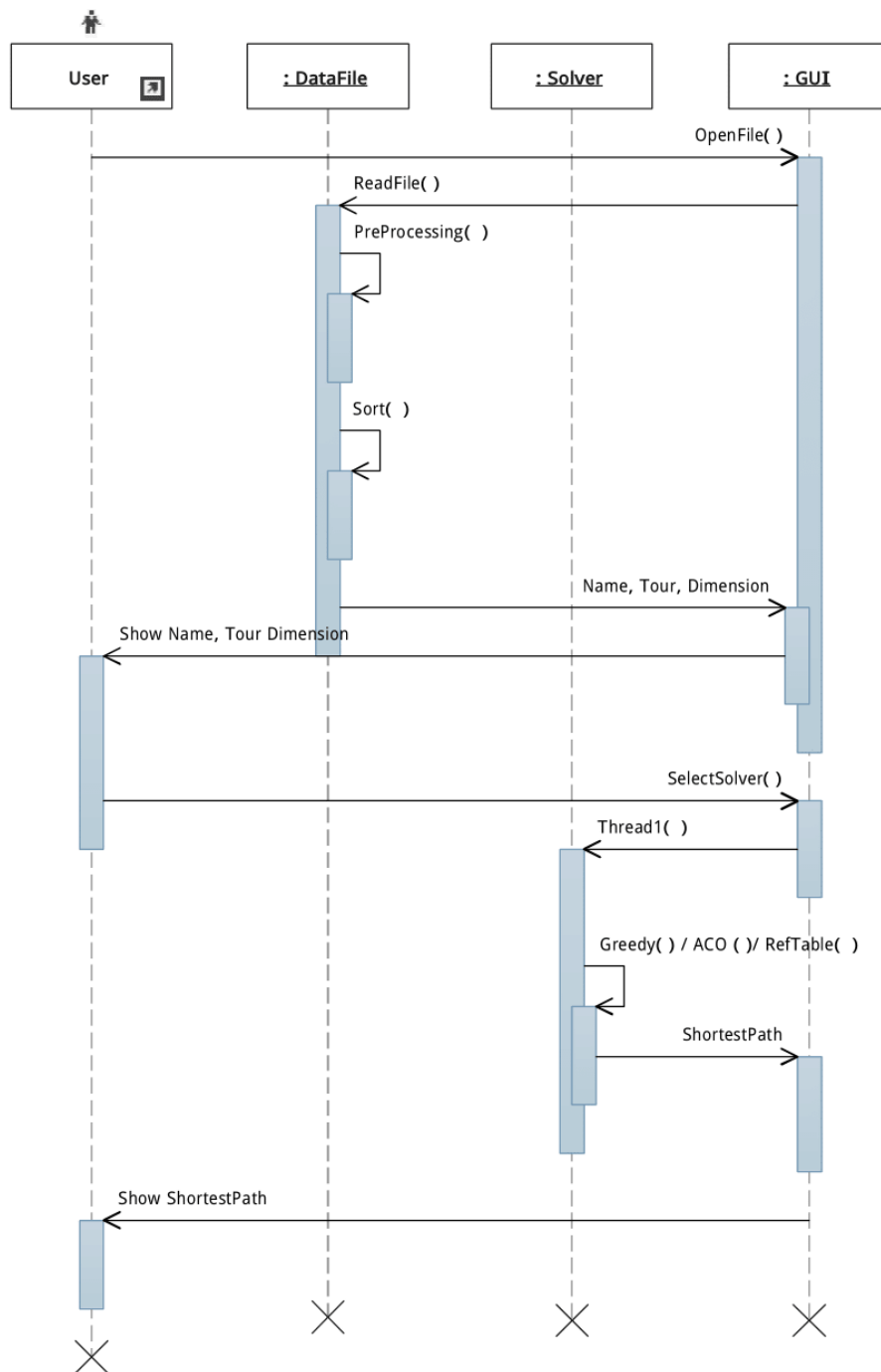


Figure 10. Design Model Sequence Diagram



Our final sequence diagram states that the user will select a file, after reading the file, the DataFiles class will perform the pre-processing on the information contained within it, it will then provide the GUI with the file name, number of cities, and optimum tour length, which the GUI will then show the user. The user will then select a solver, which will make the GUI create a new thread for that solver and run it. After the algorithm is done, the Solver will provide the GUI with the results, which the GUI will then display to the user.

The Class Diagram is now complete, including additional methods and whether attributes and methods are private or public, some new attributes have been added to fulfill the requirements for some method constructors.

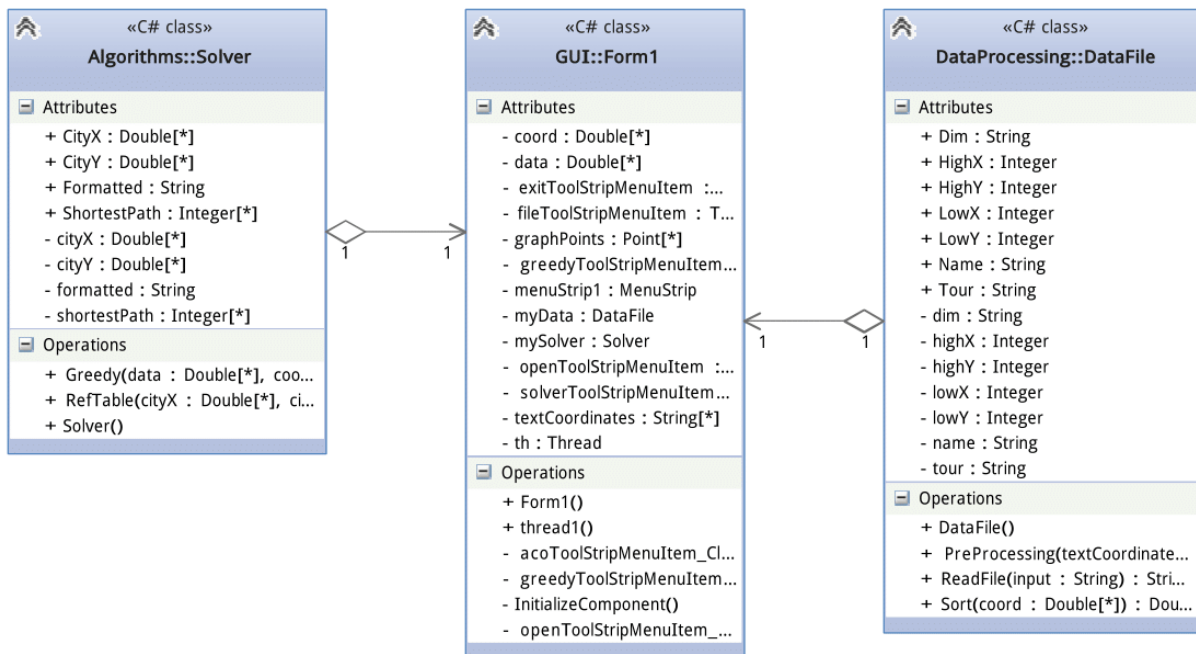


Figure 11. Design Model Class Diagram

## IV.1. Code Flow

Using this class diagram we can explain how the program works; first a file is opened via the GUI, after which it sends the file to the DataFiles class, which uses the `ReadFile()` method to read its contents using the `System.IO` namespace, it searches for the desired data by using the `LastIndexOf()` method and `IndexOf()` method on the file and looking for specific key words that are the same for all files. Then it uses the `PreProcessing()` method to build an array which lists all the cities and orders them by shortest distance using the `Sort()` method,. After it's done, it returns a double array with the coordinates to the GUI, and the GUI has access to the name, ID, and optimum tour length of the map through those variables' get and set properties. The GUI also has access to 4 other variables which are needed to scale the plotting graph and the cities' coordinates. After grabbing all of this data, the GUI displays it to the user, after which it waits for the user to select a solving algorithm. As soon as the user selects a solving

algorithm, the GUI creates a new thread through which it instantiates an object of the solver class and calls for the corresponding method.

The Greedy method works by comparing the distances array from the preprocessing, which is in order from shortest to longest distance, and another array which is the same one but not in order, it simply calculates at what distance each city is from the rest but doesn't sort them. Knowing the next shortest distance its simply a matter of comparing both arrays and seeing where they match, wherever they have the same value, that's the next closest city, if it has been visited before (controlled by a Boolean value) it's skipped and the algorithm continues, if it's not been visited before then that's the next reference point from which we compare the next closest city, and the algorithm begins anew until all cities have been visited, after which it returns to the starting point and adds all the distances traveled. This happens however many cities there are in the map, after each iteration it checks to see of the new result is shorter than the last, and that's how we can know which one was the best.

When the algorithms are finished they pass the results to the GUI, which then scales the distances again and plots them and presents them to the user. If the user wishes to load a new map file he is free to do so and the program will handle clearing the screen and stopping the thread if it's still running.

The statechart diagram needed no further revisions, no subsystems were required since it's a simple design, and there are no legacy issues since everything was done using the same piece of software.

## V. Testing

Various tests were performed in order to test the functionality of the program, the first of which was creating a test file with only 5 cities in order to test the implementation, which can be seen in Figure 12

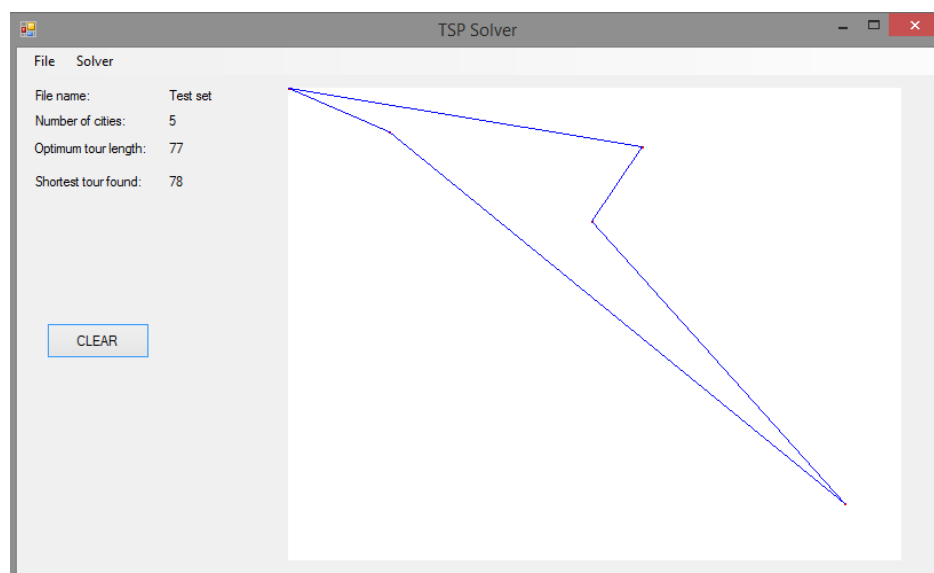


Figure 12. Test Set

Then the testing continued with various maps:

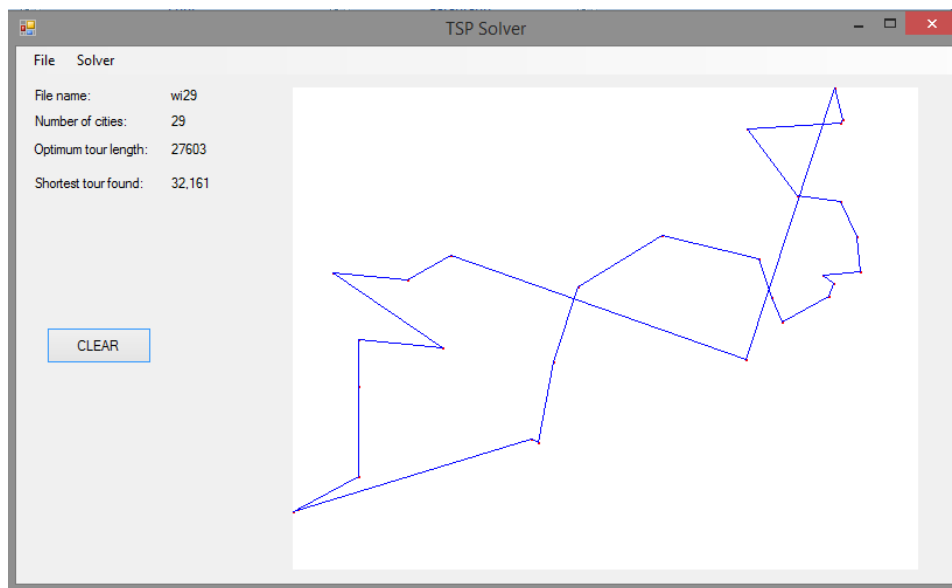


Figure 13. 29 Cities Test

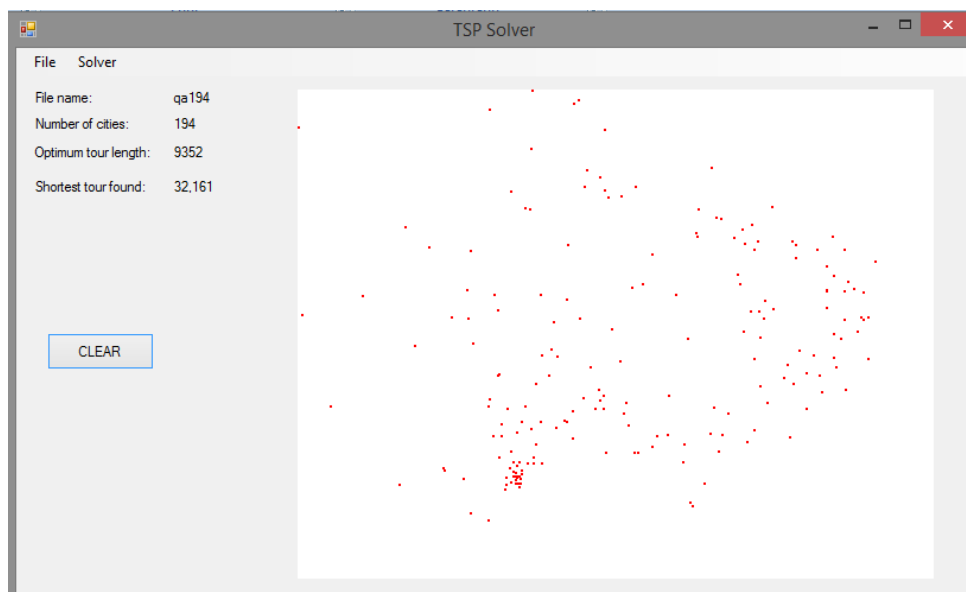


Figure 14. 194 Cities, before running solver

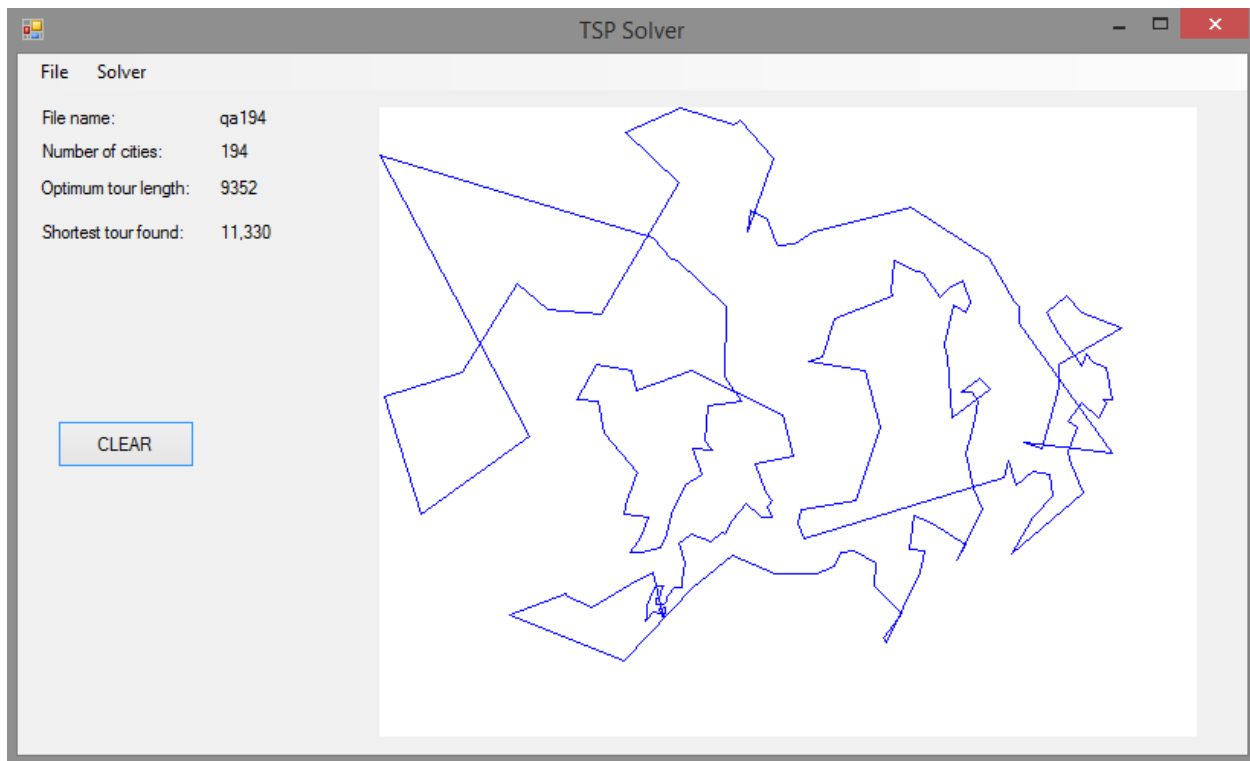


Figure 15. 194 cities, after solver

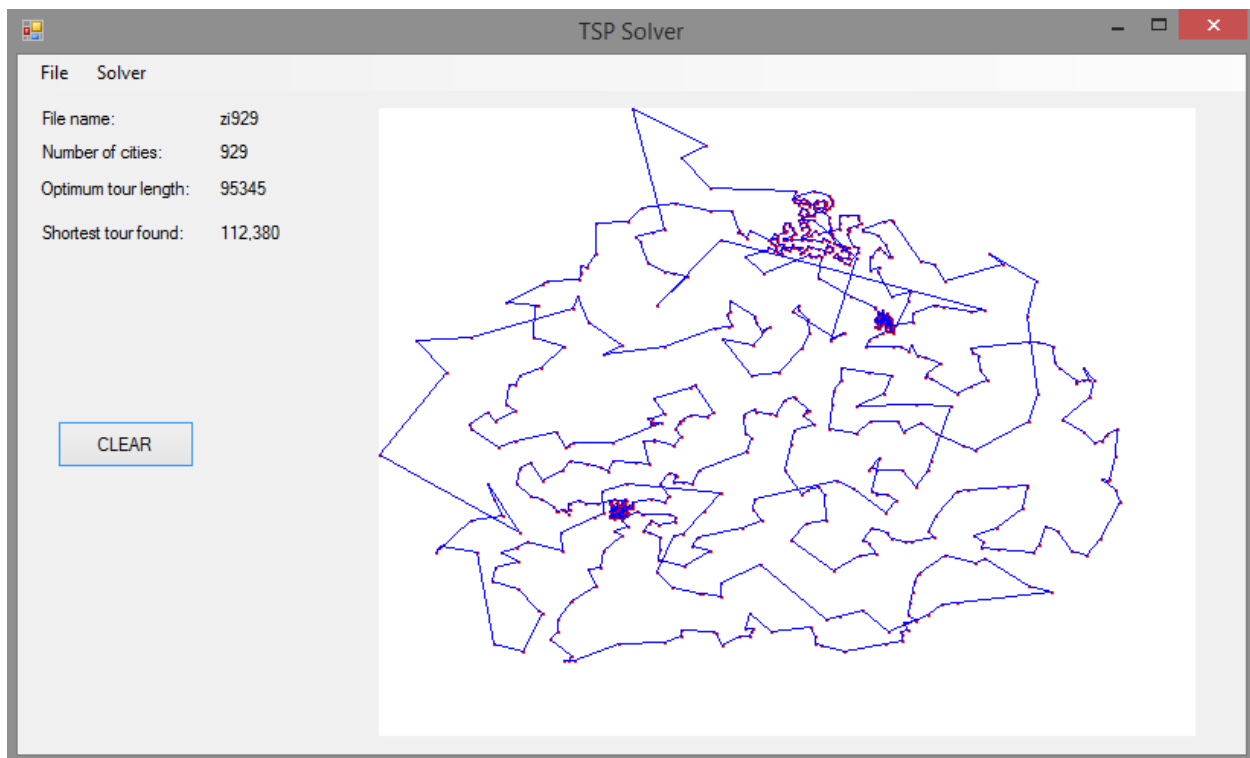


Figure 16.929 cities result

## VI. Conclusions

To further improve the program design I could have used better sorting algorithms, since the insertion method is far from the most efficient, this would make the pre-processing much better and faster. As far as the specification is concerned, unfortunately I failed to meet it, since I could not get an ACO algorithm working, due to having problems with understanding the concept and had problems implementing the pheromones, specifically the evaporation of the pheromones, I ended up with stagnation all the time, for which I had to suppress that part of the code. Obviously this would fit in as an area of opportunity to improve the program's functionality and given more time I'm sure I could implement it.

What was achieved, however, was a fully working Greedy solving program, which uses a GUI in order to display results to the user, this is something I never would have thought I could do, since all the knowledge about C programming I had was that of ANSI C, I knew nothing about methods, classes, multi-threading, GUIs, static, private and public variables. I think the progress I did manage to make is sizable considering the point at which I started. The UML for OOP was also really eye-opening, since it made me realize there's more to programming than simply starting to code right away, there's a process behind it, and it's a thinking process which is very useful both before, and especially during, coding.

In the beginning I thought this was a very ambitious assignment, thinking I could never get any part of it working, however, as soon as I started working on it I realized it was feasible and all I needed to do was keep going at it, trying to complete that next part of the program, troubleshooting, debugging, and learning along the way. I may not have completed the assignment the way I would have liked to, but everything I've learned while doing what I did manage to make is something I'm still proud of and that only make me a better programmer.

## Bibliography

- Applegate, D., Bixby, R., Chvátal, V., & Cook, W. (2007). The Travelling Salesman Problem. *Princeton University Press*. Princeton, New Jersey, United States. Retrieved from <http://press.princeton.edu/chapters/s8451.pdf>
- Bachir, B., Ali, A., & Abdellah, M. (2012). Multiobjective Optimization of an Operational Amplifier by the Ant Colony Optimisation Algorithm. *Scientific & Academic Publishing*. Retrieved from <http://article.sapub.org/10.5923.j.eee.20120204.09.html>
- Dorigo, M., & Stutzle, T. (2004). *Ant Colony Optimization*. United States: The MIT Press.
- Liu, T., Moore, A., Gray, A., & Yang, K. (n.d.). An Investigation of Practical Approximate Nearest Neighbor Algorithms. Pittsburgh, Pennsylvania, United States. Retrieved from <http://www.cs.cmu.edu/~agray/approxnn.pdf>
- Panait, L., & Luke, S. (n.d.). Ant Foraging Revisited. *George Mason University*. Fairfax, Virginia, United States. Retrieved from [http://www.cc.gatech.edu/~turk/bio\\_sim/articles/ant\\_foraging\\_revisited.pdf](http://www.cc.gatech.edu/~turk/bio_sim/articles/ant_foraging_revisited.pdf)
- Passino, K., & Liu, Y. (2000, March 30). *Swarm Intelligence: Literature Overview*. Retrieved from <http://www2.ece.ohio-state.edu/>: <http://www2.ece.ohio-state.edu/~passino/swarms.pdf>