Visualisng Ant Colony Optimisation

Final Report for CS39440 Major Project

Author: Christopher Edwards (che16@aber.ac.uk)
Supervisor: Dr. Neil MacParthalain (ncm@aber.ac.uk)

April 23, 2015
Version: 1.0 (Draft)

This report was submitted as partial fulfilment of a BSc degree in
Computer Science (G400)

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB
Wales, UK

Acknowledgements

I would like to thank my supervisor Dr Neil MacParthalain who has provided incredible help throughout the projects development. I appreciate the time he has spent with me at various times which has allowed me to develop a greater understanding the underlying algorithm behaviours. I would also like to thank Neil Taylor who has been very informative in regards to what is expected from a major project. My appreciation also goes to everyone involved with the department of Computer Science at Aberystwyth University for proving the resources necessary for the completion of a successful project such as this.

My Thanks is also expressed to my fellow final year students, especially Thomas Keogh, for spending many hours in the Delphinium over the course of the projects development enabling the countless hours spent testing and debugging much more enjoyable. Finally I would like to thank my mother Diane, father Paul and brother Michael for continued support and motivation throughout my degree.

Abstract

Ant Colony Optimisation and its variations are commonly used swarm intelligence methods, however the underlying concepts can be difficult to comprehend for people who have recently come across the subject area. The majority of existing resources either inadequate visual representations or rely on the user having some prior knowledge about the underlying behaviours. The author of this project aims to create an application for deployment in educational environments allowing for a richer, more interactive experience in regards to the teaching of Ant Colony Optimisation methods. The author has set out to achieve a full visual representation of the algorithm's execution as well as providing an intuitive user interface allowing for user defined algorithm parameters and a choice of algorithm types and modifiers.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Background

1.1 Algorithm Overview

Ant Colony Optimisation is a probabilistic technique usually used on problems which can be resolved by resolving an optimal path through a graphical representation of a given problem. Ant Colony Optimisation refers to a collection of methods and techniques which represent a specific family in swarm intelligence. Swarm intelligence "...deals with natural and artificial systems composed of many individuals that coordinate using decentralized control and self-organization. In particular, the discipline focuses on the collective behaviours that result from the local interactions of the individuals with each other and with their environment" [14]. In simpler terms each agent is simple and each agent follows a series of fairly simple rules as it performs its operations. If you increase the population size of these simple agents and allow them to communicate with each other and the world they are in then there will be an emergence of intelligence which would otherwise be unavailable to any individual agent. Ultimately each agent collectively works towards the same goal increasing the quality and appropriateness of the result.

The initial proposal for Ant Colony Optimisation came from Marco Dorigo et al. through the publication of his PhD in 1992 [6].The Algorithm is based upon the real world behaviour of ant colonies. Ants in the real world will generally always find the most optimal bath between two or more points, often described the route between their nest site and the location of the food source(s). As an Ant leaves the colony in search of a food source it begins to deposit a chemical trail (pheromone) which can be analysed by other agents in the population. The pheromone however is non consistently present once deposited, as this is the real world there are several factors which impact the concentration of the pheromone at any location once it has been deposited. Once an agent has deposited the pheromone it starts to decay overtime, and in the real world this could be accelerated by outside factors; such as adverse weather conditions. As the pheromone decays, new pheromone will be deposited at the same location by other agents in the colony also looking for a food source because of this, as more and more agents continue through their tour the locations which have the highest concentration of pheromone are generally the most traversed locations. Ultimately the locations with the highest concentration of pheromone will not only be the most frequently used, but together they will form the optimal route between the start location and the destination. This is due to the fact that the longer the route, the more susceptible pheromone levels are to decay, thus the longer paths will have less pheromone present. The pheromone levels are important because these levels are the main influence in the probabilistic function for any agent choosing its next location for any given intersection. The higher the concentration of pheromone, the greater the probability that the agent will choose this location as the next stop on its tour, however as this is probabilistic the agent may not always choose the location with the highest concentration allowing for other solutions to be sought after, allowing for a shift in the current best path.

1.2 Double Bridge Experiment

The double bridge experiment is an early experiment devised to help understand the real world behaviour of ants and their path finding capabilities. The double bridge experiment, as the name suggests involves a nest location separated from a single food source by two bridges. This experiment designed and carried out by Deneubourg and colleagues in 1989-1990 performed using real Argentine ants [12].

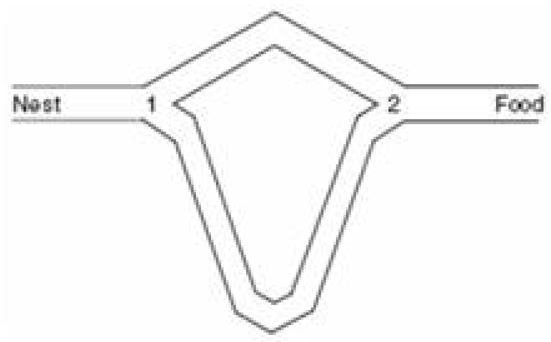


Figure 1.1: Image representing the Double Bridge Experiment. Image source [23]

Figure 1.1 represents the scenario the Argentine ant were faced with for the Double Bridge Experiment. As previously stated in section 1.1 the foraging behaviour of most ant species is dependant of communication using the pheromone deposits by each colony member during its tour. Initially there will be no pheromone trails for the ants to follow, so as the first ant approaches intersection marked “1” in figure 1.1 the probability that they choose the top path, or lower path is therefore 50%. Regardless of which path the ant chose, pheromone will now be deposited on the path it took. Now the next ant will approach the intersection marked “1” however, as there is no an existing pheromone trail this ant now no longer has an equal probability to choose either path but instead is more likely to chose the same path that the previous ant had taken. This process will continue for every ant in the nest. The pheromone will however decay. Generally speaking as the bottom path is significantly longer than the top path, the pheromone deposited on its path will more subject to this decay, and will eventually have a lower pheromone concentration than the shorter top path. Overtime this will cause more and more ants to take the top path over the bottom path due to the higher pheromone level directly impacting the probability of the ant choosing this path. The same applies to intersection marked “2”, the ants will still tend to prefer the path with the greater pheromone concentration, the fact that the ant now has food does not affect the ants choice in anyway, aside from the fact that the ants new target is the nest and no longer the food source.

1.3 Travelling Salesman Problem

One of the most common applications for Ant Colony techniques is the Travelling Salesman Problem (TSP). The TSP consists of a graph of N cities, and you must find the shortest route between each of these N cities, however each city can only be visited exactly once. Generally this N value is often fairly large for example the Berlin52 [16] is a variation of the TSP where this N value is 52. The number of possible routes between these 52 cities is incredibly large. The application of heuristic algorithms such at the Ant Colony methods enables solutions to be found within a reasonable time. One solution for the Berlin52.tsp problem is shown in figure 1.2.

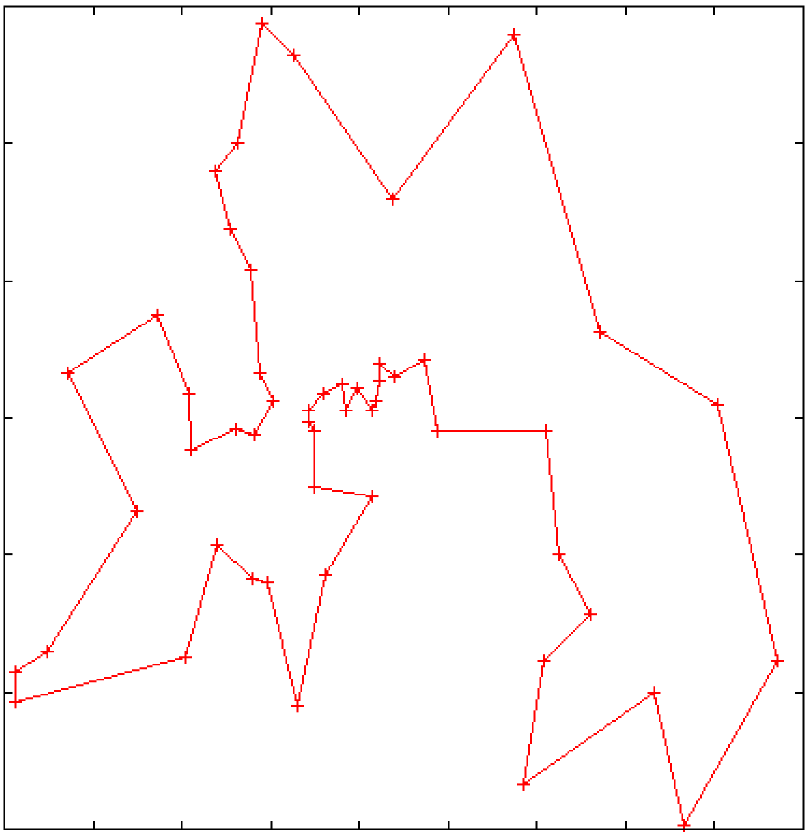


Figure 1.2: One solution to the Berlin52.tsp problem plotted using GNU Plot. Modified version from original image source [5]

The TSP is not the only problem which can be tackled using the Ant Colony family of methods. As these methods have metaheuristic properties the general behaviours and structure can be applied to several other problems such as image segmentation however this application will not cover such problems and will focus on the TSP style problem.

1.4 Ant System

The Ant System is the most basic implementation of an Ant Colony Optimisation method, because of this it provides the basis for other extensions and variations. This does not mean that the Ant System is lacklustre in performance. Due to its basic nature, the Ant System is ideal for demonstrating and teaching the behaviours of a virtual ant colony to someone, and it can be done regardless of their prior background knowledge. In this implementation there is no recollection of the best path between iterations and every agent is equal in terms of its importance in finding a solution.

1.4.1 Forumlae

Sections 1.4.1.1 and 1.4.1.2 refer to the underlying formulae which govern the agents pheromone deposits for a given edge and the probability for an agent to move to a specific location anytime it reaches an intersection.

1.4.1.1 Probability

The probability formula is as described in appendix B section 2.6.2.1. This is the function which drives the agents movements and enabled agents to generally pick the best looking path (path with the strongest pheromone concentration) whilst also allowing for agents to select other paths helping to prevent localised solutions forming.

1.4.1.2 pheromone

The pheromone function for the Ant System algorithm is as described in appendix B section 2.6.2.2. This function is the real difference between the Ant System implementation and the Elitist Ant System.

1.5 Elitist Ant System

The Elitist Ant System is the first adaptation to the initial Ant System algorithm. The Elitist Ant System is again, proposed by Marco Dorgio et al. in his 1992 PhD Thesis [6]. The main difference between the Elitist Ant System and the Ant System is the fact that the best ants for a given iteration have pheromone deposited upon their route. This means that the best x number of ants where x is an integer representing the number of elite ants will have their routes remembered across iterations allowing the fact that they performed well to persist, with the intention to improve the performance of the population as a whole as the extra pheromone on these elite paths will increase the probability that any agent traverses an elite edge (an edge that is part of an elite path).

1.5.1 Forumlae

Sections 1.5.1.1 and 1.5.1.2 refer to the underlying formulae which govern the agents pheromone deposits for a given edge and the probability for an agent to move to a specific location anytime it reaches an intersection.

1.5.1.1 Probability

The probability function for the Elitist Ant System remains the same as it is in the Ant System, see section 1.4.1.1 and appendix B section 2.6.2.2

1.5.1.2 pheromone

As stated in section 1.4.1.2 the pheromone function is the main different between the two algorithm variations. This new pheromone function takes into account that there is elite agents and enables pheromone to be deposited along the current X elite paths.

$$p_{xy}^{k} = (1-\rho)\tau_{xy}^{k} + \Delta\tau_{xy}^{k} + e\Delta\tau_{xy}^{best} \tag{1}$$

Figure 1.3: Algebraic model of the pheromone deposit function for the Elitist Ant System [11]

The majority of the formula remains the same as defined in appendix B section 2.6.2.2 however, there is the addition $e\Delta\tau_{xy}^{best}$. This is the part of the formula which is responsible for the pheromone deposit on the retained best (elite) paths currently found. xy refers to the x and y coordinate for an edge in the best path this is where pheromone will be deposited. $best$ simply donates that this edge belongs to one of the currently stored best paths. The e value is a constant, and varies between implementations. Research suggests that a good value for this weighted value, e is $\frac{1}{4} \cdot \#\ of\ nodes$ [19] however, there is evidence to support using $\#\ of\ nodes$ as the e value [13].

1.6 Excisting Solutions

There are a number of pre-existing solutions which attempt visualise Ant Colony algorithms however, the majority of these based upon the authors experience are in fact subpar in what they visualise. Rather that visualising the algorithms execution in a logical manner, which would enable the user to understand what the algorithm is doing and how the best path converges over time, more often than not the existing solutions simply show the algorithms final state leaving the user confused.

Another problem with some of the existing solutions is that the graphical user interface for the application is far too inconsistent. Inconsistent here refers to the fact that there are a number of interaction method present which can be quite daunting for a user and therefore might cause a less than satisfactory experience for example, some features may require a user to use a scroll bar to set the value, others may require direct textual input and another feature may require the user to check a checkbox. In combination with this, the interfaces themselves are usually far too crowded and often unnecessary features for the current algorithm selection. If the user has selected that they want to use a specific variation of an Ant Colony algorithm then there is no point in allowing them to be able to modify a parameter which will have no impact on the current execution. The irrelevant interface elements can cause confusion amongst the users in terms of what in fact they are modifying and how it impacts the algorithms behaviour.

One of the major problems the author faced when assessing the competition was the fact that there was rarely any visualisation of the agents themselves, this meant that you has to effectively guess which cities the agents were currently at. In addition to this as there was no visualisation of the agents there is no visualisation of the agents moving between the cities themselves thus, it was difficult to visualise the path the agents took aside from coming to your own conclusion based on the pheromone trails which were also often poorly represented.

1.7 User Interaction Methods

As discussed in section 1.6 the methods of user interaction must be superior to what is provided by the competition. This application will be authored in accordance with several preferred user interaction methods enabling a more user friednly experience for all users and not just those who are experienced with this or similar applications.

1.7.1 Law of Context

The law of context refers to the users expectation that they should only see interface controls relevant to the current object they want to control [1]. This relates to one of the fundamental problems found in competitor applications (see section 1.6). Should the user request a change of algorithm type which requires an extension to default features, these new controls will be self-contained and represented suitably so the user knows that the news dialog or interaction method is a direct result of request a change in algorithm type, allowing for a logical mapping between their selection and the new control interface.

1.7.2 Law of Feedback

The law of feedback related the ideology that every significant action has some form of informative, relative feedback associated with it [1]. This enables the users to quickly develop an understanding of what interaction control which action allowing for a more enjoyable experience. This also covers any incorrect actions performed by the user. The application will be developed in such a way that any incorrect actions will be displayed to the user in a manner that anyone can understand and provide the user with the required knowledge to resolve said issue.

1.7.3 Law of Easing

The law of easing is very important, especially for this application. This law suggests that complex actions should be segmented into simpler steps to allow the user to comprehend what they are actually doing [1]. The way this application will adopt this is that rather than specifying all of the algorithms parameters at once, each parameter will have its own method of interaction and its own series of user feedback prompts enabling any user to simply modify select parameters however they see fit, assuming the value is legal.

Chapter 2

Analysis

2.1 Key Tasks

Based upon the research undergone in chapter 1 the problem can be decomposed into several key tasks. These tasks will directly relate the requirements described in appendix A section 1.3.3.

2.1.1 Problem Representation

The problem representation is key for both creating an application which is best suited to the proposed task whilst also ensuring the underlying algorithm can correctly calculate a solution in a reasonable manner. Chapter 1 explains two stypes of problem which could be represented in this application, the Double Bridge experiment (see section 1.2) and the Travelling Salesman Problem (see section 1.3) each of these problems has its own merits and demerits.

2.1.1.1 Double Bridge Experiment

Opting to represent the problem in a similar format to the Double Bridge Expriment would enable the visualiation but more user friendly as the actual problem itself is far simpilier than the Travelling Salemen Problem. In addition to this users may relate more to this style of problem as they can map the on screen representation of the nest and food to a real world scenario, this is not easily done with the Travelling Salesman Problem. The authors feels that representing the problem in this basic form will negatively impact the applications success as having a nest location, a food source and two bridges for the agents to navigate accross can only be represented so many ways before the user becomes fully aware of the expected outcome.

Instead of maintaining the exact ideas behind the Double Bridge Experiment, the key concepts can be extracted and applied to a more complicated problem representation. The idea that every agent will start and the same nest location, and attempt to navigate to a food source is a fairly simple concept to understand, however as described in Appendix B section 2.7.1 the world can be expanded to no longer only accomodate two paths for the ants to navigate, but instead have a matrix of nodes which vastly increases the problems complexity. As the problem is no longer represented as a simple 2 path paths, it becomes exceptional more difficult to predict the agents movements which could lead to a much richer experience for the users.

2.1.1.2 Travelling Salesman Problem

Representing the problem in the form of a Travelling Salemen Problem may come across as quite daunting to new users. As discussed with the Double Bridge problem representation the idea of a nest site and a food source is easily relateable to the real world for the majoirt of people however, the Travelling Salesman problem does not boast any kind of instantly relateable real world mapping to ants and thier colonies. One of the main advantages with the use of the this representation is that the graph is a fully connected graph. As this is the case the author does not have to worry about complications such as wrapping the edges of the graph to simulate connectedness, this would be the case if the Double Bridge and the suggested modifications were to be used.

This form of representation is far more customisable thus it may be more appropriate an educational application such as this as it enables a greater degree of user customisation allowing for users to experiment with different configurations in order to understand how the algorithm operates in different scenarios.

Initially the author chose to use the suggested adaptation to the Double Bridge Experiment however, the result this produced happened to be very unsatisfactory. As this is the case the application underwent a somewhat agile re design process in order to accommodate the change in problem representation to the Travelling Salesman type of problem (more details found in REFERENCE PROCESS AND DESIGN CHAPTERS HERE TALKING ABOUT CHANGE OF REPRESENTATION.

2.1.2 Algorithm implementation

The applications success will be heavily dependent on having a working, customisable implementation of at least the basic Ant System (see section 1.4). If the algorithms implementation is flawed in anyway then is cannot be accurately visualised. As this is application has educational objectives incorrect implementation of underlying algorithm(s) would cause improper algorithm behaviours to be both taught and demonstrated thus, the quality of the implementation is paramount to the success of the application. The combination of background research and a suitable test strategy will ensure the application exhibits expected behaviours.

2.1.3 User interaction

The methods of user interaction will be key in ensuring the users actually fully utilise the functionality on offer. Using the key user interaction principles identified in section 1.7 the author will develop a simple yet effective user interface incorporating these key design concepts. The user must also be familiar with the look and feel of the interface, research will be carried out and will consist of looking at successful application of similar design then adapting any suitable design styles they provide to suit this applications needs.

2.1.4 Algorithm Variations

Given the time constraints of the project it is not a realistic to attempt to implement support for every variation of Ant Colony algorithms. It is far more realistic for the application to support few algorithm variations which will at a minimum include two different algorithm variations so users can have some insight into how different algorithms impact the overall result. Although it is unrealistic given the current development timeframe, this does not mean that supporting the majority of algorithm variations cannot be a high level objective for the application.

2.2 Objectives

The objectives listed below are generalised overall system goals. The author fully expects to have objectives which are not yet met, however these will be used as a reference point to further develop the application for future releases. The design decisions will be made with these high level objectives in mind and any design implementation which negatively impacts the applications ability to meet one or more of these objectives needs strong rationale for doing so.

- Create an application which can visualise a basic Ant System algorithm execution including the problem representation, the agents solving the problem and the optimal returned path
- Provide a suitable user interface which enables the user to modify the algorithms key parameters enabling the user to experiment with different parameter combinations
- Provide the visual representation of the algorithms execution in real time
- Allow the user to set the algorithms execution speed, enabling the user to slow the algorithm down if they want to focus on each agents movement or speed the algorithm up if they are more focused on the returned solution
- Support the majority of Ant Colony algorithm variations to enable the user to freely choose any algorithm they wish
- Support multiple problem representations, enabling the user to change the problem representation as they wish
- When using the Travelling Salesman Problem representation allow for the ability for the graph to not be fully connected to see how the algorithm returns different results
- Support the option for some paths between nodes to be weighted, these paths will cost more to travel one way than the other
- Allow configurations to be loaded from and exported to external files, enables the user to consistently apply the algorithm to the same problem

2.3 Requirements

These requirements are seen by the author as essential features that the application must provide. The requirements defined in appendix A section 1.3.3 describe the immediate goals for the application however, there is some overlap between the overall objectives described in section 2.2 and the functional requirements. In addition to this each functional requirement defined does not have equal importance, some of the requirements are indeed critical and must be completed at the earliest convenience. The evaluation of these requirements is described in appendix A table A.2 which shows a summary of all requirements and their dependencies.

2.4 Desireable, non-essential Features

The features mentioned in this section are seen by the author as non-essential, however, if implemented these features would significantly improve the applications performance. The implementation of these features will only be considered, if and only if the requirements mentioned in section appendix A, section 1.3.3 have been met to a significant degree of accuracy. The application will however be designed in a way that will allow modification and maintenance to be less problematic, therefore the author feels there should be no reason why these features could not be implemented if there enough time during the development process to do so.

- Enable the user to switch between automated solving and an iterative step-by-step process, enabling a greater teaching potential.
- Upon completion or stoppage, a summary or report should be displayed to the user, enabling the user to perform analytics based on the contents of these reports.
- Allow the ability for the application to generate graphs which are not fully connected to visualise how the algorithm deals with such complications.

The author realises that these features are outside of the projects general scope. The author also feels that although they maybe outside of the projects scope the implementation of these features is by no means out of the realms of possibility. The intentions and efforts of the author will reflect the functional requirements being a top priority.

Chapter 3

Process

This project is largest project the author has solely been responsible for however, this does not define the project as a large project. The author is working alone and does not have a dedicated team as assistance. This combined with the fact that the time frame which the project has to be completed in has lead the author to come to the conclusion that there is no single Process Model that would be entirely suited for him and this project. The lack of one specific process gives the author the temptation to fall back on process at all however, the lack of any rational structure is vastly inappropriate for this project. Rather than using one single process, the author feels that is can modify existing processes in order to create a process which captures the necessary principles required in order to create a logical structure which can be applied effectively to this project. Agile methodologies are often used in small development teams, specifically eXtreme Programming (XP). XP places heavy importance of quality and responding to customer demands through the use of multiple iterations and a simple design process (compared to planned design). However, small development teams are using XP as a scapegoat for not adhering to a specific process or methodology. XP is still very much focused around a team of people working are there is constant deliberation about XP and its adaptation to single developer project such as this [4]. However, there are XP principles which are well suited to this style of project for example, there is less emphasis on a heavy design process allowing for a more flexible development process which has been vital for this project in particular as the proposed design had to be significantly changed during development (more details can be found in section 4). The author feels that in a solo project, the ability to be flexible and independent of any concrete design is key to a successful project. As the author has created a brief design specification individually without the help of a specialised team (see appendix B) there is the possibility that there are some key elements missing which may not be noticed until the implementation stage. XP flexibility effective makes this a non-issue during development as it accounts for the projects features to change, if the author adopted a strict plan based methodology then this maybe an issue.

As stated the process used is a hybrid process containing relevant ideologies from other software development processes. In conjunction with XP principles discussed above, the author has decided that there has to be some planned process involved in the projects development. The idea behind using a planned approach as the main underlying process is because the author feels that when working on solo projects it is easy to get lost in terms of what goals still need to be achieved. This planned process wont be as strict as it would be if a planned approach was the sole process being used for this project, instead the design and requirements processes will be much shorter than usual and will only cover the main requirements and overall system interactions.

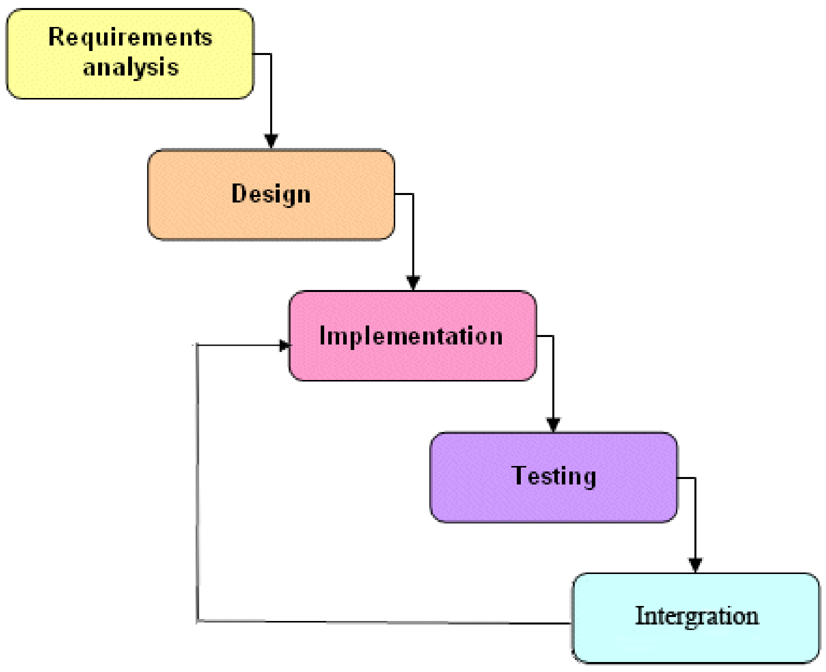


Figure 3.1: The varation of the waterfall model used for solo development in this project. Modified from source image [21]

Figure 3.1 represents the variation of the waterfall model used as part of the hybrid process in this project. The general waterfall structure has remained the same, the process starts off with a problem analysis stage which is reflected in appendix A and chapter 2. With the introduction of the XP principles, the design stage is reduce in its complexity. This is with the intent to reduce how concrete the design is and also reduce the amount of time spent on the formal design of the application. The main adaptation comes from the approach to implementation and testing. As in introduction of XP has enabled the author to be more flexible in the implementation process if a new feature or refactoring is required, rather than going back to the design stage, this new process allows the author to implement the change or refactored code, test the changes and integrate the changes into the current working application. This saved tremendous amounts of time during the development process. Each change is handled in small sections rather than implementing several changes at once, each change will be broken down into logical section which will each in turn go through the implementation and testing process. This process continues until the requirements define in appendix A have been adhered to. This hybrid process is based on the work by Tom Blanchard [3], the process is very effective for use in solo projects.

Chapter 4

Design

This chapter describes the design rationale in relation the current system. This chapter will cover the overall system archtiecture, user interface designs as well psuedo code reprsentations of the main, non-trivial algorithms.

Section 3 states how the author has a somewhat flexible approach allowing for adaptations to the initial design stated in appendix B. This section explores how the process has enabled the significant change of design to be implemented easily and without severe complication. The proposed design in appendix B and the final structure of the system are very different however, at an abstract level the key components still exist they are just represented differently within the current architecture.

4.1 Language

The author has identified that the choice of language is an essential decision which must be made early in the projects development cycle. Appendix B, section 2.2 denotes the language decision process the author went through and ultimately states the final language choice.

The use of an appropriate language became even more important due to the fact that the design has undergone significant modifications since the initial proposed design. If the author had selected a language which happened to be unfamiliar but seemed good at the time, then the changes made to the initial design may have been much more complicated to implement. The author is very familiar with Java, more specifically Java 7 (Java 1.7) and above thus, implementing new features or refactoring existing systems is fairly familiar territory. The initial language choice rationale (appendix B, section 2.2) did not factor in the potential for radical design change however, if it did the outcome would not change and the existing rationale would still be applicable to the project. In the authors opinion Java is the most appropriate language for this project.

4.2 Development Tools

4.2.1 Development Enviroment

4.2.1.1 Command Line Tools

The author considered using the combination of a simple text editor which in this case would have been Atom [10] to write the code, and compilation of such written code using the command line and the default complier through the use of the $javac$ command. Atom is a free, open source hackable text editor provided by GitHub and provides syntax highlighting support for numerous language, including Java which is the language of choice for this project. This syntax highlighting is especially useful and makes the writing of the source code slightly easier, as is becomes much simpler to track the start and termination points for specific code blocks. However, as Atom does not have built in Java compilation abilities, the compilation process can become somewhat complicated.

Using this approach, the Java source files composed using Atom must be complied using the default Java complier, provided by the language specification. The problem with this method of compilation is that error detection and correction can become a very tedious process. If any complication errors are present after the $javac$ command has been executed, the trace presented in the terminal window can sometimes be quite difficult to parse for the error message, and tracking down the error in the source files could be a difficult task. For smaller projects, this method of complication is perfectly suitable and can be effectively managed. However, this project as discussed in section 3 this is a rather large project for the author to undertake. As this is the case, the combination of size of the applications source code and manual complication leads the author to believe this method of writing and compiling code is not only inefficient, but also inappropriate for this project and could potentially hinder development.

4.2.1.2 Intergrated Development Environment

The use of an Integrated Development Environment (IDE) can help to alleviate the problems that can come from using a writing and complication process such as that defined in section 4.2.1.1. There are many IDE's which support Java, however the author has preference is the use of the Eclipse IDE for Java Developers [9]. The Eclipse IDE provides a multitude of useful features as part of their default package.

One of the main features which Eclipse provides is automatic building of the probjects source. This is a major benefit when developing as you can simply write your code and run the built project without having the unnecessary complication of switching between several applications to achieve the same task. Also, as there is automatic building, any compilation errors will be clearly highlighted using a system which is familiar to most computer users. If there are any miss-spellings of method or variables names for example, Eclipse will underline these in red to show there is a clear problem with this exact line of code. This is similar to the default scheme provided by most word processors, enabling a logical mapping between this red underlined line of code and the fact that there is an error present there.

Another useful feature provided by the Eclipse IDE is the auto completion of variables and method names. This has sped up the development process for the author as there was no need to constantly look at the API's or other project source files for method names, the author could simply type the Object of interest's identifier and see a list of all methods and variables associated with such Object. This is not possible to do if the author used an approach similar to that discussed in section 4.2.1.1.

In addition to the built in compilation feature, Eclipse also provides an integrated debugging toolkit. The Eclipse debugger enables easy creation and removal of break points, as well as all the usual features you would expect from a debugger such as, step through and step over, exploring the various contents of different variables and objects. These features are also provided by command line debugging tools, however the author is far more comfortable using a graphical interface to debug the application as it is much earlier to access the features of the debugger using this approach.

Eclipse also has built in Junit support. This project will use the JUnit library as part of the testing process (more details see 4.2.3.1). Similar to how the IDE makes the writing of the applications code easier, the support of test suites such as JUnit makes the test process much easier. The code responsible for the tests will also have access to the features provided by the IDE as mentioned above, this includes in place builds and auto completion. In addition to this you can run the tests inside the IDE and get accurate feedback on how many test ran, how many failed and why these tests failed. The author uses this to identify failed tests to be easily identified and repaired. Eclipse also supports a variety of other languages and features but these are not relevant to this project currently.

4.2.2 Support Tools

4.2.2.1 Maven

Maven is tool used for building and managing projects, specifically Java-based projects [2]. The Maven framework provides a lot of useful utilities for developing successful Java-based projects and ensuring that such projects adhere to certain standards. The Maven framework aims to provide convention, over configuration. If there are multiple projects which have many different dependencies, some of which could be external Jar files, then Maven can allow other projects to make use of these dependencies. As a result, if a new project is decided there is less time spent configuring as the dependencies are already presented using this Maven framework. However, the author is working alone, and is not expecting to have any other projects which will share dependencies with this one, thus the author has to weight up if this framework would actually help the development process.

Maven provides some useful testing capabilities. When testing using the Maven framework, tests are still written using the Junit libraries (see 4.2.3.1). However, the Maven framework provides some additional features which the author feels could be of use. Maven, will give the author a Unit test report should one be required which will cover numerous details, most important a coverage report will be produced. This enables the author to assess how well the unit testing has been done. Although this test report is a very useful feature, the Maven is not a necessity nor is it complication free.

The main complication that the author has with use the of Maven for this project is that fact that as this is generally a small project, Maven and its features would not be fully utilised but the hassle of configuring the framework would still be present. The initial project configuration would be time consuming, given that the author is inexperienced with the framework and this time could be spent on the actual project development. There are plugins which enable Maven to be used with an IDE such as Eclipse, the author feels that Maven would not be appropriate for use here, however the test reports would be extremely useful.

4.2.3 GitHub

The author has seen the use of GitHub as an essential part of the development process. GitHub has enabled the author have strict version control throughout the development process. As this is the case, the author has been able to store a local repository, which will exist as the working directory for the applications development whilst also allowing a working copy, with the current most up-to-date fully working project code to be stored safely online using GitHub. There has been numerous times throughout the development lifecycle where the author has had to revert back to a previous version of the applications code, GitHub has allowed this process to be more hassle free. As it is so simple to manage different versions of the application, each representing a different collection of working or non-working features the author has been able to develop in confidence as he knows that any mistakes or experiments are not very costly as there is always a backup version stored at the GitHub repository location.

Without the use of GitHub (or another version control system) the author would have found it difficult to progress the application as expected. If there was no form of version control, then the mistakes the author made or external factors which then rendered the current development code unusable would be a far more disastrous. There was a time during development where the authors working directory became corrupt. This was easily resolved by simply retrieving the code stored in the projects GitHub repository, however if GitHub was not being used here, then there would have been a high possibility than the author would have lost all progress on the application.

4.2.3.1 JUnit

JUnit is an open source framework for unit testing Java applications. The Junit libraries provide a multitude of features that allow for a more simplistic approach to testing. Numerous other test frameworks are available however, the author is very experience with JUnit and feels that the features provided by this framework are more than suitable for the unit testing of this application.

The author feels that a testing framework is essential for efficient and accurate unit tests for any application. If the author had neglected to use a testing framework such as Junit and manually tested the application using a looking manually for abnormal output (LMFAO) then the quality and coverage of the unit testing would reduce significantly. If the author used a manual approach such as this which relied on outputs then the author could potentially have a situation where the code in outputting the correct result, but the inner workings are expressing incorrect behaviours which cant be manually checked. This means that you could ship a product which is assumed to be functioning as expected then suddenly, the application is significantly misbehaving. In addition to this it is difficult to automate manual testing as the author would need to check the outputs himself. A testing framework such as JUnit not only allows for the tests to be fully automated, the tests also become scalable. Every code change would need to be re-evaluated against the existing unit tests, which is easily done if you have a series of test which can be executed automatically. If a manual approach was used, then it becomes a near impossibility that every change can be tested as it would simply be too much to check as the addition of this new code has scaled the number of application inputs and outputs as well the conditions in the codes logic.

4.3 Overall Architecture

The process used during this applications development has enabled the author to modify the proposed overall architecture significantly. This has enabled the author to add additional features which were not additionally planned, this includes the addition of extra algorithm types and modifiers as well as enabling the potential heavy refactoring of code into more logical methods and classes. The new application architecture is much more complex that the initial proposed design, as a result each package will be represented as its own set of diagrams, and then an overall representation showing the interactions between these packages will be explained. In all figures below, getter and setter methods are emitted but are assumed to be present. The packages referebced in this section will take the form of $che16.dcs.aber.ac.uk.xxx$ where xxx is the corresponding package name.

4.3.1 Controller

The Controller package has undergone several changes since the initial design of the system as defined in appendix B, section 2.3.3.3. The general ideologies have however remained the same, the ranges of features provided by the application as a whole has increased thus, the author saw these modifications the Control package contents as a necessity in order correctly control these new features whilst also adhering to the existing Model-View-Controller framework.

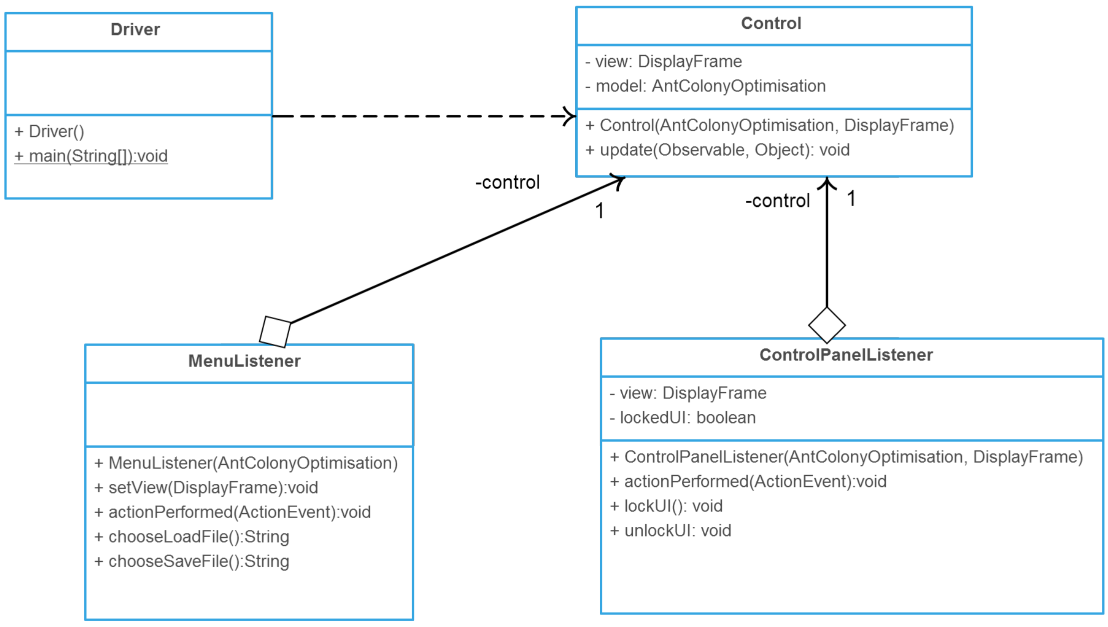

Driver
+ Driver()
+ main(String[]):void
Control
- view: DisplayFrame
- model: AntColonyOptimisation
+ Control(AntColonyOptimisation, DisplayFrame)
+ update(Observable, Object): void
-control
1
-control
1
MenuListener
+ MenuListener(AntColonyOptimisation)
+ setView(DisplayFrame):void
+ actionPerformed(ActionEvent):void
+ chooseLoadFile():String
+ chooseSaveFile():String
ControlPanelListener
- view: DisplayFrame
- lockedUI: boolean
+ ControlPanelListener(AntColonyOptimisation, DisplayFrame)
+ actionPerformed(ActionEvent):void
+ lockUI(): void
+ unlockUI: void

Figure 4.1: The contents of the Control package show in standard UML Class diagram notation

4.3.1.1 Class Descriptions

The **Driver** Class remains largely unchanged from its initial proposed design (see appendix B, section 2.4.1.1) however, it now interacts with the other components in a slightly different manner. The Driver Class remains as the entry point for the application, however, the Driver has the additional responsibilities of instantiating the Control instance, as well as handling the correct associations needed. The general purpose of the Driver Class is still to ensure the system components are correctly instantiated, and holds no references to any instances of any Object.

The **Control** as represented in figure 4.6 is designed to observe the model and notify the view should the state of the model change significantly. In addition to this, the required instanced of MenuListener and ControlPanelListener are instantiated in this Class, and a reference to this Control Object is maintained the created instances of said Objects. The Control Class is an additional feature not present in the initial design, this is because as represented in figure B.2 the DisplayFrame initially observed the model, this has since been modified. This Class has the potential to be extended to support several other Observable Objects should this functionality be needed by the author in the future.

An instance of the **MenuListener** is used to listen the JMenuBar present in the DisplayFrame. This is designed to enable a collective way to manage all actions represented by the items contained in the JMenuBar. The alternative approach is to give each element in the JMenuBar its own Action Listener. This is far from efficient and increases both the codes complexity, and reduces overall maintainability of the application. The author decided to use a dedicate Object such as this, to listen to the menu and perform appropriate actions. This Class maintains an instance of the Control Class, this enables the instance of this Class to have access to the model and view instances stored in the Control instance enabling a simple way for the JMenuBar to have access to necessary functions.

A **ControlPanelListener** instance is designed to be a dedicated ActionListener for the ControlPanel (see section 4.3.3.1) instance. Two separate ActionListeners are present in this application, this is because the author wanted to have dedicated listeners for each component, rather than having a combined Object representing this Class and the MenuListener Class. The author feels that this is appropriate as any behaviour modifications or changes the Object which will be listened to becomes much simpler to accomplish if relevant behaviours are extracted into logical modules such as this design exhibits.

4.3.2 View

The View package serves the same purpose as initial designed however, there has been a singnificant amount of refactoring applies to the initial design to enable a more effective graphical user interface to be created. The author has also added several new Classes into this package to represent additional views which were not initially visioned, but during development the author recognised that these new views were essential.

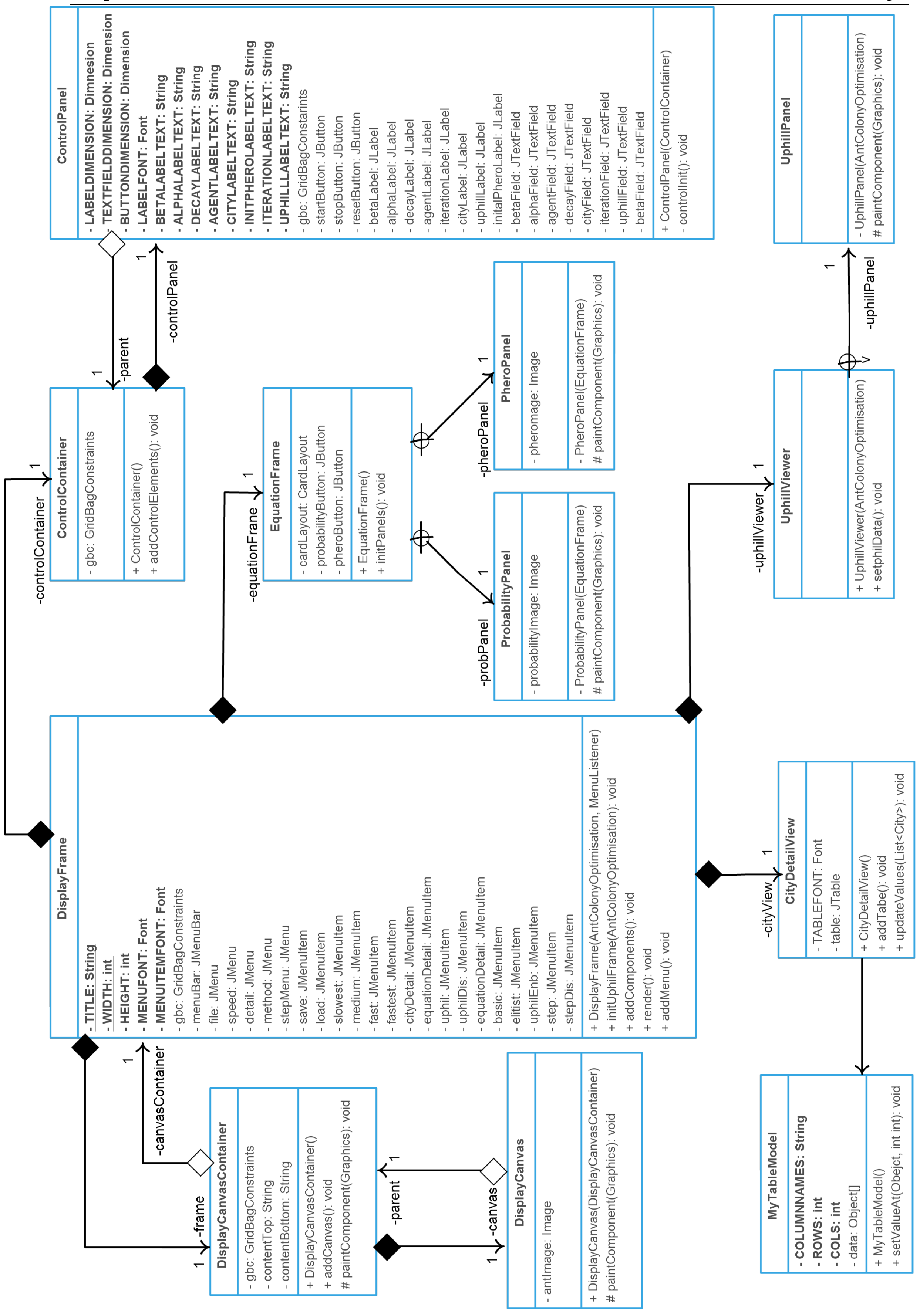


Figure 4.2: The contents of the View package show in standard UML Class diagram notation

4.3.2.1 Class Descriptions

Initial the **DisplayFrame** was designed to simply represent the highest level container which houses the remaining graphical user interface elements. As figure B.4 in appendix B shows the initial design for this class happened to be very simplistic and lightweight. The author decided to include an additional user interface elements which was not originally planned. This addition happened to be JMenuBar, which was contained within this DisplayFrame instance. As this is the case, the DisplayFrame Class grew it size as it now encapsulates the JMenuBar and the menu elements of said JMenuBar as well as defining constants such as font styles. Although the size of this Class has grew, the complexity and functionality remains unchanged, it remains as the highest level container for the graphical user interface and control the instantiation of such elements.

The **DisplayCanvasContainer** is the result of renaming the DisplayPanel Class in the initial design (see Figure B.4, appendix B). This Class is designed as a container to the DisplayCanvas instance, this functionality is as designed and the author has not modified its behaviour.

An instance of the **DisplayCanvas** Class is used to actually visualise the algorithms state of execution to the application user. This will be the component that is painted during the algorithms execution using the $paintComponent$ method which it inherits from its JFrame super Class. As the functionality provided by this Class is very simplistic, the current implementation remains unchanged from the initial proposed design (see Figure B.4, appendix B).

The addition of the **CityDetailView** is a simple JFrame container which is used to display a JTable with data representing the number of agents currently at each city in the current World. The JFrame displayed by this Class can be toggled between visible and invisible through user interaction with the respective JMenuBar item.

The structure of the table represented visually by the CityDetailView, is modelled by the **MyTableModel** Class. Extracting the tables structure into a separate Class enables easier modification of the structure as well as reducing the coupling between the view and the table itself.

Similar to the DisplayCanvasContainer Class, the **ControlContainer** Class is used to contain the user input elements in their own separate location. This is a result of refactoring the UserInputPanel Class in the initial design (see Figure B.4, appendix B) into this container and the ControlPanel Class. The main purpose for this Class is to allow more a more diverse range of positioning utilities to become available allowing for an easily modifiable interface.

The **ControlPanel** Class used to contain the interface elements which directly relate to the creation and modification of the problem and world. This includes housing the text fields and labels required to enable the user to customise the algorithm parameters, as well as providing a simple means to start and stop the execution using a simple JButton approach. This Class is essentially an extended version of the UserInputPanel Class in the initial design (see Figure B.4, appendix B), aside from the fact that more interface elements have been added, the functionality is the same.

The **EquationFrame** Class an extension of the JFrame Class which is used to display an graphic explaining the workings and functions of the underlying algorithms. As this is an educational application. The Class has no other functionality aside from the providing a means of displaying such graphics which is itself, contained within the ProbabilityPanel and PheroPanel Classes.

Both **ProbabilityPanel** and **PheroPanel** are nested Classes inside the EquationFrame Class.

These Classes are extensions of the JPanel Class and are used to contain different graphics which will be used to represent information relevant to the underlying probability and pheromone functions respectively.

the **UphillViewer** Class is another new addition which was not perceived in the initial design. This Class is a sub Class of JFrame and is used to house an UphillPanel instance. This JFrame can be togged between visible and invisible with relevant user interaction with the JMenuBar contained in the DisplayFrame.

An instance of the **UphillPanel** Class is used to display information about the current status of the uphill routes for the current algorithms execution. This is an extension the JPanel Class, enabling the uphill route data to be painted to the component using the inherited $paintComponent$ method. This is a nested Class inside the UphillView Class as the author felt a this was the most appropriate way to represent such a Class as no other Class uses or needs the knowledge of this Class.

4.3.3 Model

The elements contained in the initial proposed designed represented in Figure B.4, appendix B have undergone significant refactoring which has produce a vastly different structure. During the development process the athour found the the intial design happened to be inaquate for representing the aglorithm in a suitable manner, as it lacked necessary components and detail.

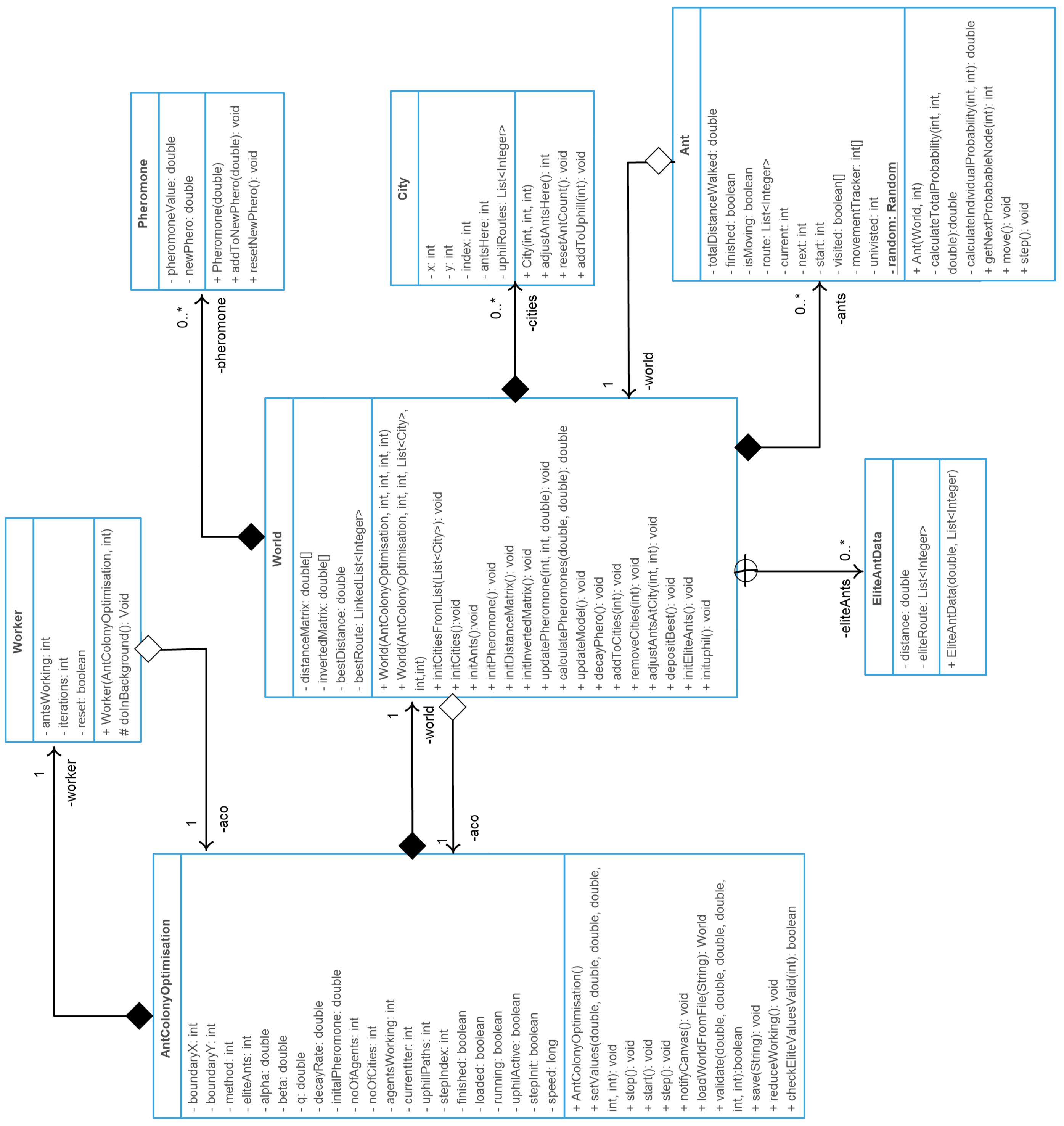


Figure 4.3: The contents of the Model package show in standard UML Class diagram notation

4.3.3.1 Class Descriptions

The initial concept behind the **AntColonyOptimisation** Class is that this servers as the main control point and data centre for the algorithms representation. This Class is used to store all the necessary parameter values and is also responsible for in instantiation of the Word representation and therefore, indirectly the graph. This Class also controls external package access to the Model package and its data. This largely remains the same as designed, however, there are slight modification to accommodate the extra functionality added.

The **Worker** Class was something that wasnt initially planned. As discussed in REFERENCE IMPLEMENTATION OF SWING WORK there was a necessity for a way to control the algorithms execution without interfering with the performance of the system or host machine. This Class is an extension of the SwingWorker Class and uses the inherited $doInBackground$ perform the algorithms execution in a suitable manner.

The **World** Class is used to model the problem representation and the environment which the agents will be deployed during the algorithms execution. This Class houses all the data relevant to such representations including a List of all Ant and City Object as well as having basic data structures which provide an effective representation of the pheromone concentrations for every edge within the graph. This Class also handles the manipulation of pheromones on such edges. This Class is generally the same as proposed in Figure B.4, appendix B excluding the additional feature support.

A **City** Object is used to represent a node in the current problem representation graph. A list of these City Object is maintained in the current World instance, and iterated through during both the solving and painting processes.

A **Pheromone** Object is used to model the pheromone concentrations for any given edge. A two-dimensional array of these Objects is maintained in the current World instance, and these Objects are constantly manipulated during the pheromone deposit and decay operations.

An **Ant** Object is used to represent an agent which will be deployed in the current environment in order to solve the current problem. The World instance will maintain a list of all current Ant Objects. Each Ant is able to move and deposit pheromone accordingly, this these Objects have sufficient variables and methods to enable this functionality. The initial design suggested the use of an AntInterface, however as the author has decided that there will only be one type of Ant Class, he has deemed the Interface to be unnecessary. Generally, this Class is as designed in Figure B.4, appendix B.

The **EliteAntData** is a nested Class inside the World Class which is used to store the current elite routes. This feature is only necessary if the user has selected ElitistAnts as the current algorithm type. This is required as if there are numerous iterations, the Ant Objects get reset each iteration and this is a lightweight way to store the elite ants for the course of the algorithm execution, as storing just the distances and routes of the best ants has less overheads when compared to storing whole Ant Objects.

4.3.4 Utils

During development the author decided that there were methods that were being repeated implemented in numerous places, in order to prevent code duplication and promote reusability the utils package was implemented. This is a very simple package, containing one Class, however, the contents of this Class don't belong in any other package individually thus, have been implemented separately.

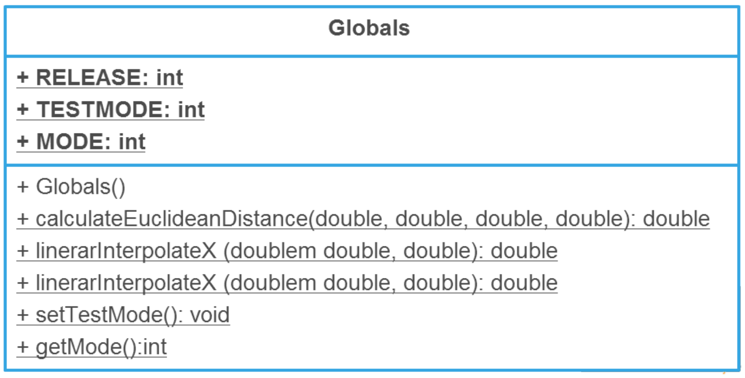

Globals
+ RELEASE: int
+ TESTMODE: int
+ MODE: int
+ Globals()
+ calculateEuclideanDistance(double, double, double, double): double
+ linerarInterpolateX (doublem double, double): double
+ linerarInterpolateX (doublem double, double): double
+ setTestMode(): void
+ getMode():int

Figure 4.4: The contents of the Utils package show in standard UML Class diagram notation

Figure 4.4 represents the single Class contents of the Utils package. This Globals Class contains several publically avliable static variables and methods enabling application wide access. This enables the multiple packages which rely on the results of the features present to still have access whilst also reducing the amount of repeated code in the application. In addition to this, this Class also enables the execution mode to be switched from release, to test mode. When in test mode the error messages require no user response enabling the automated test process to complete more easily.

4.3.5 System Interactions

The different packages and their contents have been described above, however the interactions between packages must be carefully designed to ensure correct implementation and of the Model-View-Controller design pattern.

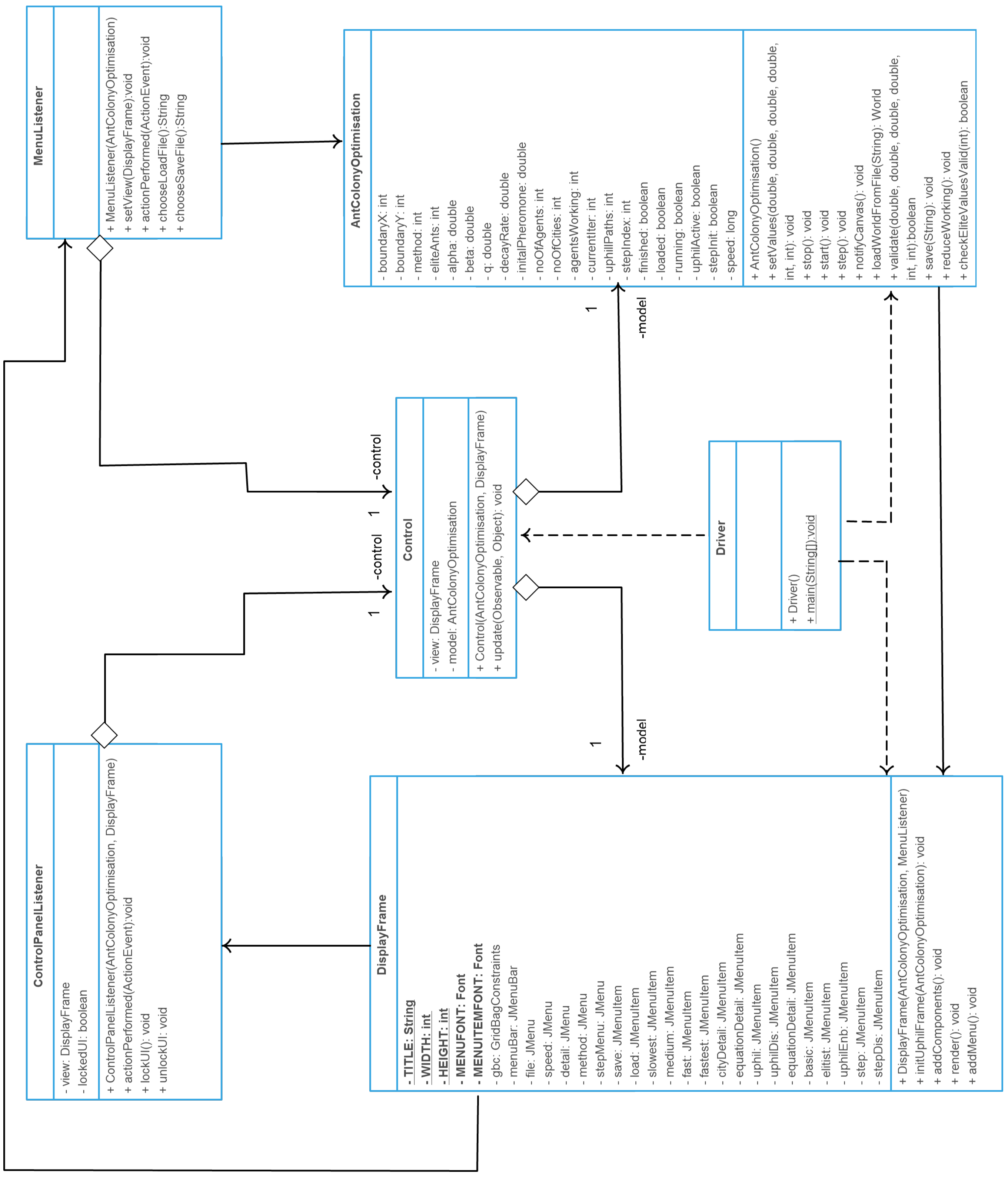


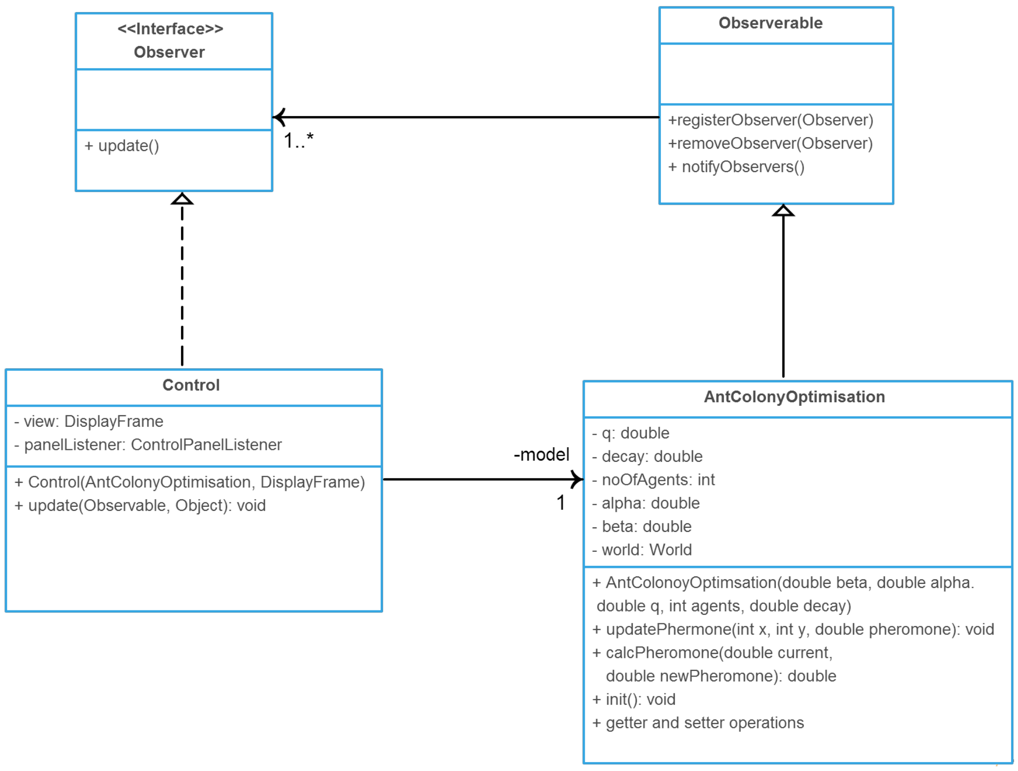
Figure 4.5: The proposed manner of package interaction expressed in standard UML Class diagram notation.

Figure 4.5 demonstrates how the entry to the view and Model packages is governed by the instances of the DisplayFrame and AntConolonyOptimisation Classes respectively. This keeps the interaction between packages simple, and enables sensitive data to be hidden, or modified easily without having to change the way the components of the system interact. This design adheres to the Model-View-Controller principles as the Controller package and its contents govern the interactions between the Model and View. The view, has no reference to the model however, at runtime an instance of the model is passed to the View in order for the painting of necessary components to take place. As there is no explicit link between the model and view, these packages can be easily modified or substituted in order to change the current representation, there is no coupling between the contents of these packages.

4.4 Design Patterns

4.4.1 Model-View-Controller

The current design still maintains and adheres to the Model-View-Controller(MVC) principles discussed in Appendix B, section 2.3.1.1 however, the complexity of the different packages has changed both in terms of internal package structure and features provided. As designed, the author has implements an MVC compliant architecture is through the use of the Observer and Oberavle relationship using the corresponding default Java Classes. The general conecpt is implemented as defined in appendix B, section 2.3.2 slight modification have been made to the Classes representing the different components of the Observer and Observable concept.



<<Interface>>
Observer
+ update()
1..*
Observerable
+registerObserver(Observer)
+removeObserver(Observer)
+ notifyObservers()
Control
- view: DisplayFrame
- panelListener: ControlPanelListener
+ Control(AntColonyOptimisation, DisplayFrame)
+ update(Observable, Object): void
-model
1
AntColonyOptimisation
- q: double
- decay: double
- noOfAgents: int
- alpha: double
- beta: double
- world: World
+ AntColonoyOptimsation(double beta, double alpha.
double q, int agents, double decay)
+ updatePhermone(int x, int y, double pheromone): void
+ calcPheromone(double current,
double newPheromone): double
+ init(): void
+ getter and setter operations

Figure 4.6: Implementation of the Observer and Observable Design pattern

Figure 4.6 demonstartes how the Control Class will implement the Observer interface, which will listen to the Observable Object and determine correct actions based upon the source of update. The AntColonyOptimisation Class is as defined in section 4.3.3.1 is the Object which is in fact Observable, and extends the Observable super Class. As the AntColonyOptimisation instance is now a subclass of Observable, it now has access to key methods such as $registerObserver()$ which allows the athor to assign the Control instance as the desiginated Observer and $notifyOberservers()$ which enables the author to dictate when updates are published, this enables the Control instance to process update notifications and notify the relevant view elements that the state of the model has changed, enabling the seperation of the view and model.

4.4.2 Singleton

Initially the author proposed that the user interface elements would implement the Singleton design pattern [7], an example this proposed design can be seen in appendix B, section 2.3.2.1 however, the Singleton pattern is not present in the current system architecture. The author initially implemented the main elements from the view package (see 4.3.2.1) this caused several complications for the author. As the instance of the Singleton Object is globally accessible, difficulties arose during the application debugging. This issue happened to be more prevalent when extending the functionality provided by each of the Singleton Classes, for example the DisplayFrame Class now handles contains a wider range of functionality than initial planned. Not only did the Singleton instance make the addition of these extended features more complicated than the author intended, debugging these new features became more difficult as the global access made it harder to track the source of the bug itself. This is due the fact that it is easy to accidently interact and modify a global variable or instance, often the source of these bugs were not where the author expected and were often due to an incorrect interaction with these new extensions in a package which should not be able to access them. This was the main reason the author decided to withdraw the use of the Singleton pattern as the author felt it was much simpler to not have global access to these instances and features.

Robert Martin created idea of the Single Responsibility Principle (SRP) in 1990 [18]. The general concept behind the SRP is that each logical module in the software should have one reason to change or model one specific responsibility rather than a collection of unrelated features or functions. If a module has been extended to support multiple unrelated features, the SRP states that the unrelated features should be extracted into relevant modules so that each module maintains its one responsibility. The author feels that implementing the Singleton in the proposed manner (see appendix B, section 2.3.2.1) violated the underlying concepts of the SRP. As the Singleton Class is responsible for tracking and instantiation of its one allowed instance and the functionality which the module presents. This Class now has more than one responsibility and thus breaks the SRP. The author believes the SRP should be regarded highly during development in order to produce maintainable and extendable software, thus he has opted for adhering to the SRP over the Singleton implementations. The author experimented with the idea of having extracting the functionality of the singleton Classes and the tracking of the instance of such Classes into separate system modules. Once this is done each module will have one responsibility thus, adhering to the SRP.

Overall, the author sees the implementation of the Singleton Design pattern as more of an anti-pattern. As this is the case and to enable easier modification and future extension the implementation of any Singleton Class as initially designed has been removed. Solutions the above problems could be refactored in, but this is seen as unnecessary complexity by the author and has therefore also been omitted.

4.5 User Interface

The user interface elements will be designed with the three law of user interaction dicussed in section 1.7 as a top prioirty. The interfaces will be consistent in theme and styling which will provide application wide consistency for the user.

4.5.1 Main Display

The design for the main display, which refers the general view the user will be presented and interact with remains unchanged from the initial proposal as represented in figure B.5. There has been modifications to this design during the implementation phase however, these the authors flexible process (see section 3) and the fact that these changes were fairly trivial allowed the author to implement them without any prior design, section 5.1 displays the implemention of this design and said changes.

4.5.2 Uphill Viewer

The uphill viewer was not an element of the user interface which was initially planned, however the addition to implement the ability for a path to represent uphill terrain called for a view to summarize which of these paths are in fact subject to this uphill modification during the given problem. This interface is a result of the contents of the UphillViewer Class (see section 4.3.2.1).

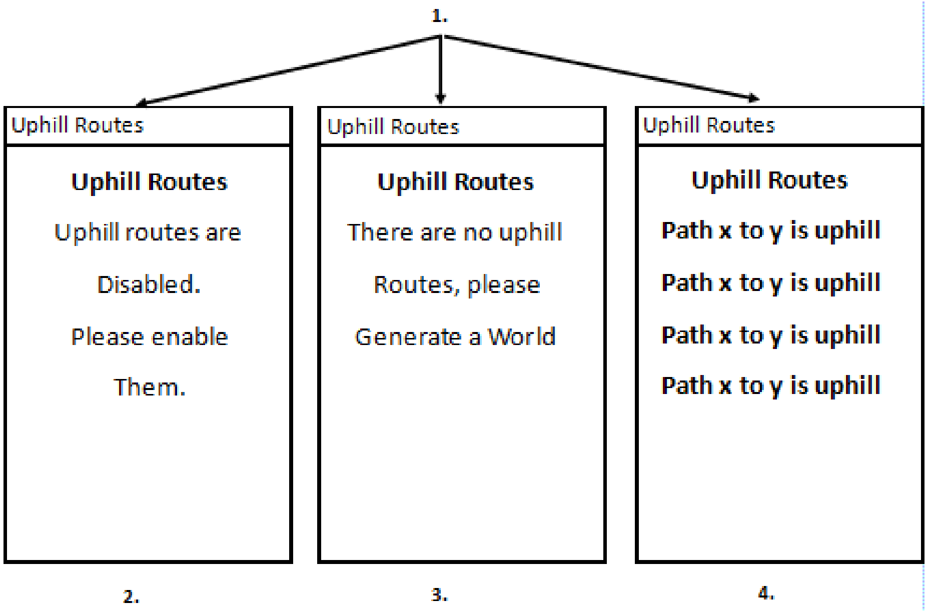


Figure 4.7: Abstract proposal for the uphill viewer interface representing the different states possible at runtime.

1. in figure 4.7 is used to demonstrate that each possible state for the uphill viewer will be represented using the same high level container. **2.** in figure 4.7 is used to show the default state and content for the uphill viewer interface. This state and content is shown to the user if the uphill routes is current disabled for this problem, and prompts the user to enable them. **3.** in figure 4.7 is shown to the user if uphill routes are enabled, but the current world is yet to be generated. **4.** in figure 4.7 is shown to the user when both uphill routes are enabled for the current problem and they have been generated. The x and y values in this figure will be replaced with indexes of valid cities to enable to the user to understand which routes are uphill.

4.5.3 City Detail Viewer

The City Detail Viewer is used to summarize how many agents are at each City for the current problem. This view was not initially designed however, the author felt that this was a necessary addition and therefore designed a rough outline to how this interface is to be presented.

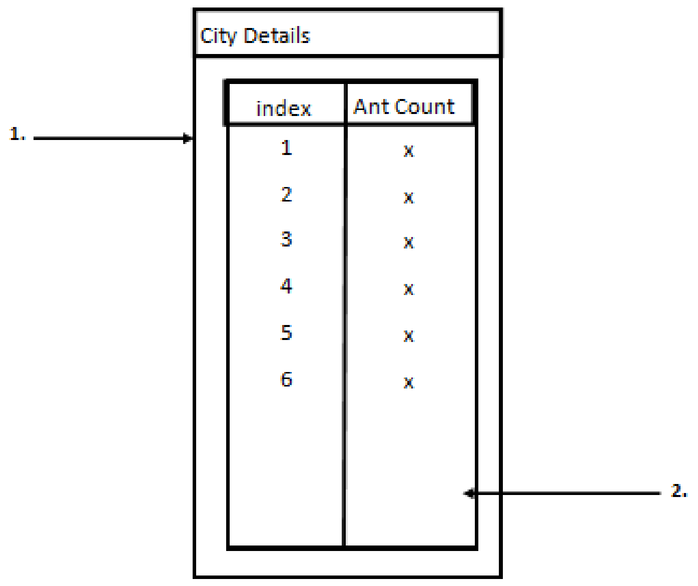


Figure 4.8: Abstract proposal for the city detail interface.

1. in figure 4.8 represents how this view is contained in a separate container to the main display. This enables the user to move this view around as desired in order to customise the current view to their tastes. **2.** in figure 4.8 shows how the contents of the CityDetailView will be implemented (see section 4.3.2.1). The number of rows will directly relate to the number of City Objects in the current problem. The x values will be substituted for the correct number of ants at the corresponding city index.

4.5.4 Equation Viewer

The Equation Viewer is used to explain the key, underlying functions present in the algorithms execution. This view was not initially designed however, the author felt that this was a necessary addition and therefore designed a rough outline to how this interface is to be presented.

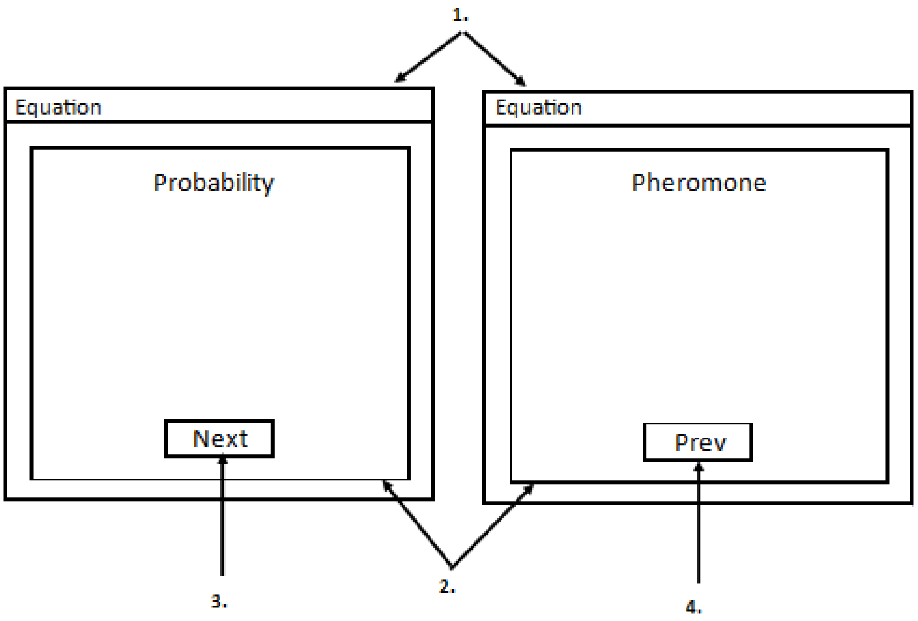


Figure 4.9: Abstract proposal for the equation viewer interface.

1. in 4.9 represents how this view is contained in a separate container to the main display. This enables the user to move this view around as desired in order to customise the current view to their tastes. **2.** in 4.9 demonstrates how there will one shared panel which will be used to display the content related to both the pheromone and probability equations. The content of the container signified by **2.** will be substituted for the correct content when the user interacts with either one of the buttons signified by **3.** and **4.**. If the user interacts with the button represented by **3.** then the content of the container will switch and will now represent details about the pheromone equation. When button **4.** is pressed by the user, the content of the container will now reflect the information stored about the probability equation. The author has designed this such that any additional equations can be implemented in the same way using simple button navigation without the need of significant extensions to the existing framework.

4.5.5 Error Feedback

The interfaces responsible for error feedback have been carefully designed so that the author can effectively inform the user of the error, the cause of the error, and solution of such error as well as enabling the interfaces to be abstract in such a manner so the same interface can be used to represent several different related error interfaces.

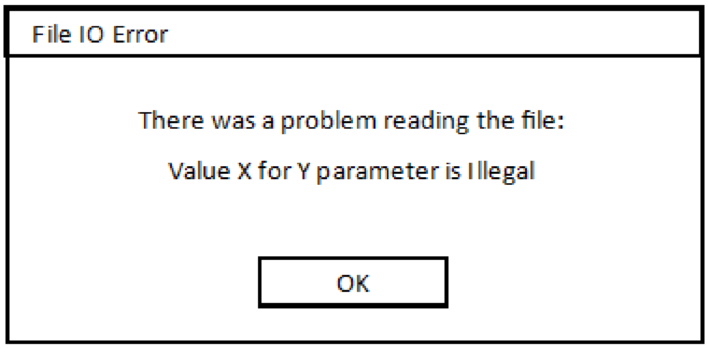
4.5.5.1 Parameter Errors

These errors are produced when a user has defined a value for an algorithm parameter which is deemed to be illegal. This illegal value could mean that the value the user has selected is outside of the scope of accepted values, or it may mean that the user has specified an illegal type for this parameter. An illegal type refers to the user potentially specifying an incorrect type for a parameter such as entering a double value where an integer is required.

The design for these error messages remains as is in section 2.5.2, appendix B. This design is simple and effective and provides the user with everything they need to know about the errors source and solution. This interface can be re used for any parameter related error as the only modification would be the change of the interfaces content, there would be no need to redefine a new interface.

4.5.5.2 File IO Errors

The ability for a use to load of save a problem configuration to a selected file is a new addition to the system. The design of these error interfaces is based off of the design for the parameter error interfaces described in 4.5.5.1.


File IO Error

There was a problem reading the file:

Value X for Y parameter is Illegal

Figure 4.10: Design of the File IO error feedback interface.

Similar to the parameter error interface, this interface as described in figure 4.10 will provide the user with concise feedback as to how and why the file IO process did not complete as expected. From the error message displayed the user should be able to deduce their own solution to the problem. As there are numerous problems which could arise with file IO, the author decided to omit providing a solution to the user. Instead the erroneous is flagged to the user enabling them to locate the problem and resolve it. The x value in figure 4.10 will be replaced with the problematic value whereas the y value will correspond the parameter that is the problem.

4.6 Algorithms

This section covers the astract implementation of the key algorithms used for both modelling the algorithms execution and the visualisation of such process.

4.6.1 General Overview

The general algorithm remains largely unchanged from the initially proposed algorithm in section 2.6, appendix B. The author had amended this design slightly to factor in the change of problem representation. As the TSP is the default problem representation the initial ideas behind ants collecting food and returning to the nest have been replaced by conditionals reflecting if an agent has visited every City or not.

Algorithm 1 Pseudo-code for Ant System implementation

```
1: Initiate AntColonyOptimisation with defined parameters
2: if !$parameters\ are\ legal$ then
3:     Dispaly error message to user
4:     $return$
5: Initiate World with algorithm parameters
6: Initiate $Nodes$ and graph
7: Initiate *pheromone* values
8: Initiate $Agents$
9: while !$all\ agents\ finished$ do
10:     for all Agents do
11:         while !visited all $City$ locations do
12:             Calculate next move using probabilistic function
13:             Add moved point to Agent's memory
14:             Calculate and deposit pheromone on the path
15:             Update the View
16:         end while
17:     end for
18: if *local best solution* $<$ *global best solution* then
19:     $globalbest = local\ best\ solution$
20: end if
21: end while
22: output global best solution
```

The pseudo code has however, remains largely unchanged from the initial design represented in algorithm 5, appendix B. There has been the modification of the conditional statement represented by line 11 in algorithm 1. This change reflects the change of problem representation so that agents now visit all City locations, rather than focussing on a nest and food style situation.

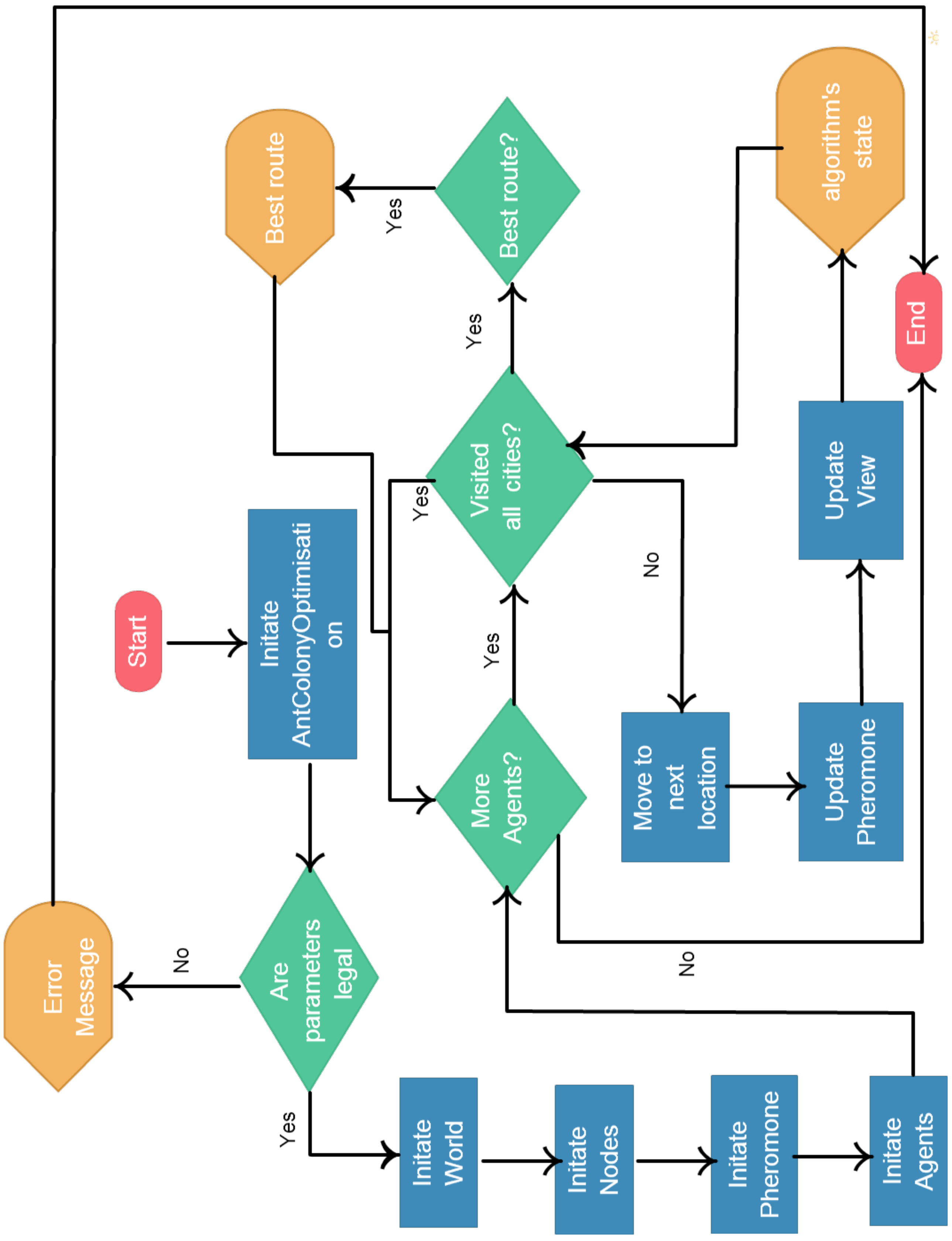


Figure 4.11: Flow diagram representation of the algorithm described in algorithm 1

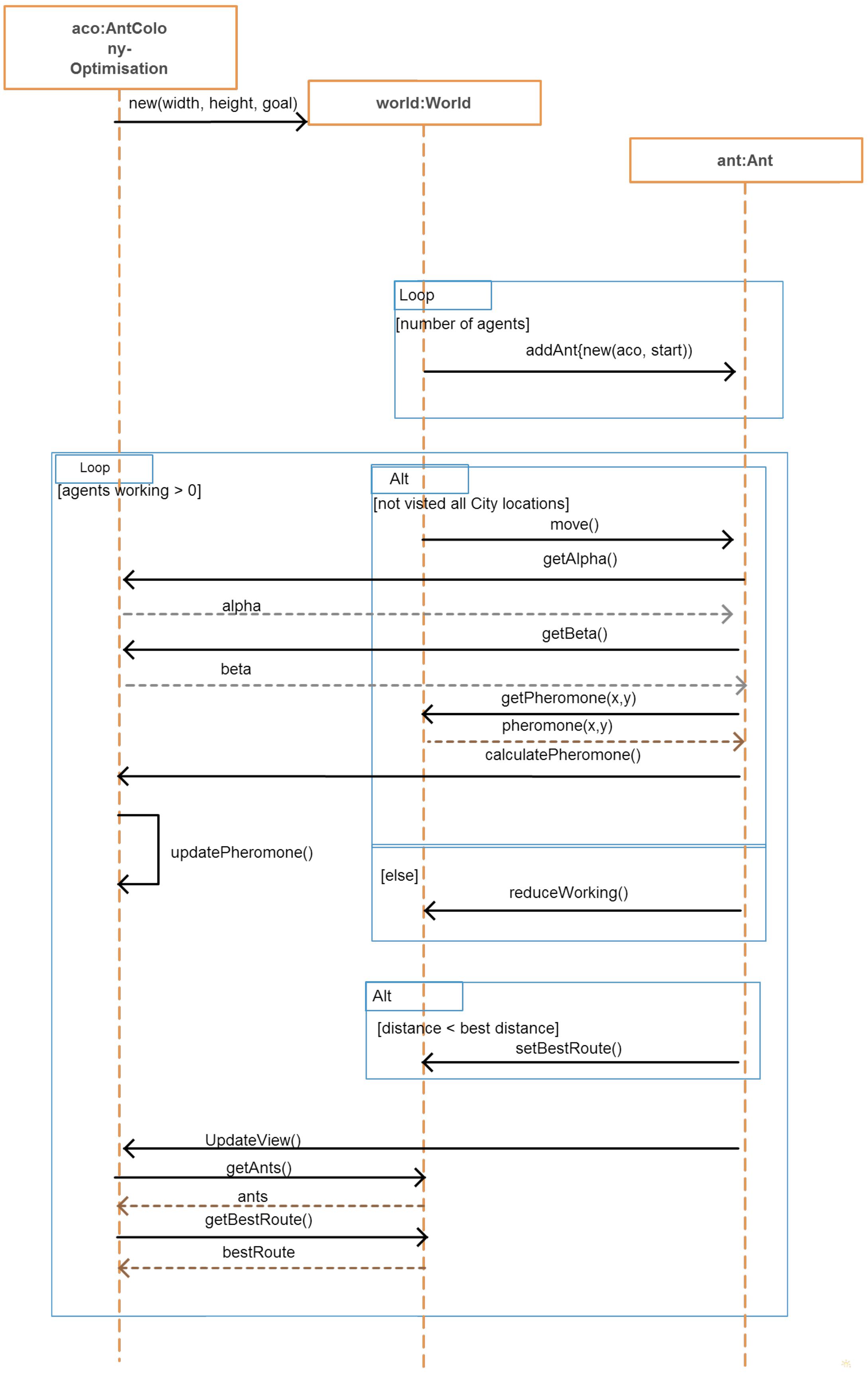


Figure 4.12: Abstract overview of system interactions based on Algorithm 1

The sequence diagram described in figure 4.12 demonstrates at a high level how the algorithm will be executed. This differs slightly from the initial design (figure B.8) in order to correctly model the change of problem representation.

4.6.2 Formulae

4.6.2.1 Probability

The probability function is defined in section 2.6.2.1 appendix B. The design for the probability algorithm reamins unchanged form the pseudo code represented in algorithm 6. The result of this algorithm is in fact, the probability assoicated with the agent moving to a specific location.

4.6.2.2 Pheromone

The probability function is defined in section 2.6.2.2 appendix B. The design for the probability algorithm remains unchanged form the pseudo code represented in algorithm 7. This is the function which is constantly updated by the agents movements in order to correctly model the pheromone update process.

4.6.3 Elitist Ants

Elitist Ants were not initially planned as a feature provided by this application, however the author decided that sufficient time was present to produce a brief design for the elitist algorithm extension to allow for a smooth implementation of such design.

4.6.3.1 Overview

The general premise for this Elitist Ant algorithm is defined in section 1.5. The probability function will remain the same as defined in algorithm 1 whereas the general behaviour of the system and the pheromone deposit function will differ slightly to that of the above proposal. The general system interactions will be the same as shown in figure 4.12

Algorithm 2 Pseudo-code for Elitist Ant System implementation

```
1:  Initiate AntColonyOptimisation with defined parameters
2:  if !$parameters\ are\ legal$ then
3:      Dispaly error message to user
4:      $return$
5:  Initiate World with algorithm parameters
6:  Initiate $Nodes$ and graph
7:  Initiate *pheromone* values
8:  Initiate $Agents$
9:  while !$all\ agents\ finished$ do
10:     for all Agents do
11:         while !visited all $City$ locations do
12:             Calculate next move using probabilistic function
13:             Add moved point to Agent's memory
14:             Calculate and deposit pheromone on the path
15:             Update the View
16:         if total stored $EliteAnts$ ¡ total defined $EliteAnts$ then
17:             add this ant to elite ants
18:         else
19:             for all $EliteAnts$ do
20:                 find the $EliteAnt$ with the worst route
21:                 if worst $EliteAnt$ route is worse than this $Ant$ route then
22:                     replace worst $EliteAnt$ with this $ant$
23:                 end if
24:             end for
25:         end if
26:
27:         end while
28:     end for
29:     end for
30: end while
31: if *local best solution* < *global best solution* then
32:     $globalbest = local\ best\ solution$
33: end if
34: end while
35: output global best solution
```

The difference between aglorithm 2 and algorithm 1 is manipulation of EliteAnts which is defined abtractly in lines 16 to 23. These lines enable the algorithm to constantly ensure that only the best (Elite) x number of ants are stored across iterations to ensure than these elite ants correctly have pheromone deposited on their paths. In order to do this, every time an ant has completed certain checks are made to see if this ant is eligible to become one of the x elite ants. An Ant is deemed eligible if the current elite ant collection is not fully complete or if this ant has a better route than the worst performing elite Ant. If the later of these two conditions is met then the worst performing elite Ant is swapped with the current Ant. This difference can also be observed in figure 4.13 which gives an even more abstract overview of the Elitist Ant System.

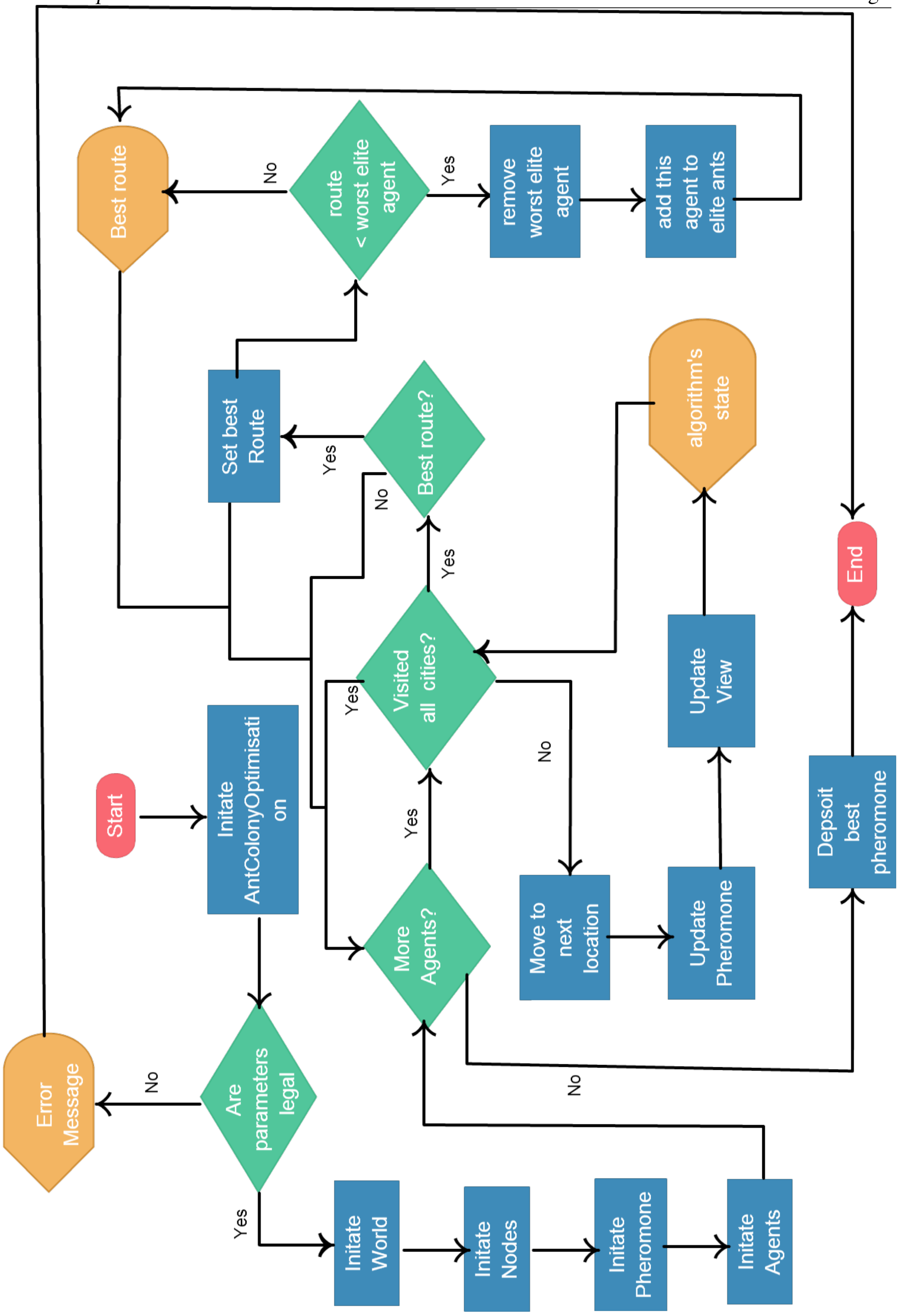


Figure 4.13: Flow diagram representation of the agloirthm described in algorithm 2

4.6.3.2 Despositing Best Pheromone

As described in section 1.5, the Elitist Ant System has a slightly different pheromone function to that of the one described in the Basic Ant System in 2.6.2.2. This modified pheromone function is mostly the same as in the Basic Ant System however, the currently stored elite routes must be factored in.

Algorithm 3 Pseudo-code for the Elitist Ant System pheromone function

```
1: define the $e$ value $= e*$ no of $cities$
2: for all stored $EliteAnts$ do
3:     initialise index $i = 0$
4:     $currentBest = EliteAnt\ path$
5:     for all $Cities$ along $currentBest$ do
6:         get the $City\ index$ at $currentBest[i]$
7:         if $i + 1 < currentBest\ size$ then
8:             get the $Cityindex$ at $currentBest[i + i]$
9:             add $e$ amount of $new\ pheromone$ to $edge[i][i + 1]$
10:        end if
11:        $i + +$
12:    end for
13: end for
```

Algorithm 3 demonstrates how the current elite ants impact the pheromone concentrations across iterations. In general, this will enable the population of agents to converge towards a solution at a faster rate than the Basic Ant System as the increased amount of pheromone on edges used in the current best paths will be greater thus, ants will have a greater probability of traversing such edges.

4.6.4 Rendering the Algorithm

Design and development of the algorithm responsible for visualising the algorithms current state to the user is one of the more important algorithm in the application as the visualisation process is key to the success of the project.

Algorithm 4 Pseudo-code for rendering of the algorithms execution

```
1: retrieve latest version of the model
2: if model !$null$ then
3:     for all $Cities$ in $model.getWorld().getCities()$ do
4:         $drawOval(City.X, City.Y, width, height)$
5:         if algorithm !$finished$ then
6:             for $City$ do
7:                 for all $Cities$ do
8:                     set opacity based on pheromone value at $edge[city.index][cities.index]$
9:                     draw a line from $City_{xy}$ to every element in $Cities_{xy}$
10:                end for
11:            end for
12:        end if
13:    end for
14:    for all $agents$ do
15:        for all $cities$ do
16:            if $agent\ location == City.index$ then
17:                draw agent at $City.x, City.Y$
18:                $break$
19:            end if
20:        end for
21:    end for
22:    if $bestdistance > 0$ then
23:        $best\ route = model.getBestRoute()$
24:        for $i = 0$ untill $i < best\ route.size()$ do
25:            if $i + 1 < \ best\ route.size()$ then
26:                set colour as red
27:                $drawLine(Cities[i].x, cities[i].y, cities[i+1].x, cities[i+1].y)$
28:            end if
29:        end for
30:    end if
31: end if
```

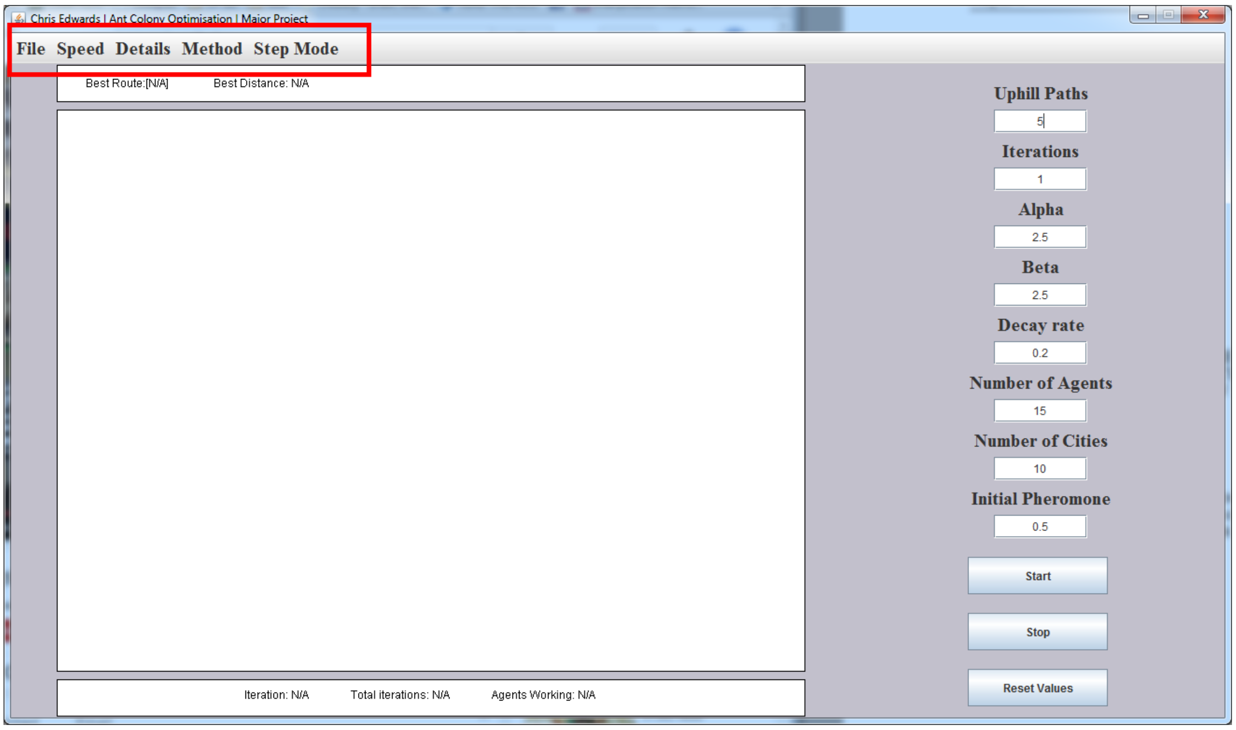
The pseudo code represented in algorithm 4 defines the design of the general concept on displaying the algorithms details to the user. The author acknowledges that this is not the most efficient or elegant design and this is used a starting point for the implementation of such algorithm and is by no means a concrete implementation. Lines 3 to 13 are used to represent the visualising of the graph. Drawing a line from every City to every other City will simulate a fully connected graph of cities to the user, the opacity of these lines will be controlled and will be directly related to the pheromone concentration for the corresponding edge. Lines 22 to 30 is the way in which the current best path will be rendered to the user. The author has designed this so that multiple pairs of lines are drawn which will ultimately form one continuous line representing the best path. Each line will have a start and end location, the end location of each line will become the starting location of the next line. This is done by iterating through the pairs indexes of the cities in the best route untill there is no longer a valid pair of locations to draw a line between. Once this point is reached the line is complete.

Chapter 5

Implementation

5.1 User interface

The main display represents the general interface is displayed to the user. This is where the algorithms visual representation will be presented, as well as providing the location of the user interaction elements enabling modification of the algorithms parameters. This interface is a result of the contents of the DisplayFrame Class and its contents. The design for this view is almost as described in section 2.5.1, appendix B, however, there has been a slight modification to this proposed design.



Best Route:[N/A] Best Distance: N/A

Uphill Paths: 5
Iterations: 1
Alpha: 2.5
Beta: 2.5
Decay rate: 0.2
Number of Agents: 15
Number of Cities: 10
Initial Pheromone: 0.5

Iteration: N/A Total iterations: N/A Agents Working: N/A

Figure 5.1: Implementation of the proposed user interface. The contents of the red polygon highlights the additional features not present in the initial design.

The additional features highlighted by the red polygon in figure 5.1 represent the features which were not initially designed. These features are represented in a separate location to the control panel (right hand side of figure 5.1) as there is no logical connection between the menu bar features, and the modification of the algorithm parameters. The author opted to use a menu bar to control the access to these features as the vast majority of users will recognise what a menu bar is, and understand how to interact with such elements.

The different elements contained in this menu bar relate to general system interactions. The File option enables the user to either load, or save a configuration to a specified file. The Speed option enables the user to change the Thread speed, which will directly change the algorithms rate of execution. The Details menu is used to control access to any additional views. The views which can be access here are; the Uphill Viewer (section 4.5.2), The City Detail View (section 4.5.3) and the Equation Viewer (section 4.5.4). In addition to the access of these extra views, the Detail menu also enables the user to disable and enable uphill routes REFERENCE UPHILL ROUTES GENERATION ALGORITHM PSEUDO CODE to be generated for current problem. The Method option enables the user to select the current algorithm type from a list of implement algorithm types. Currently the author has implemented a Basic Ant System and an Elitist Ant system so the user can switch between these algorithm types. The Step Mode menu enables the user to enable or disable the step mode functionality. When enabled, step mode will allow the user to step through the algorithms execution at their own pace without the application automatically solving the problem.

5.2 Data Structures

5.2.1 Pheromone and Distance Matracies

5.2.1.1 Pheromone

A Pheromone Object is used to represent the pheromone concentration for a given edge in the current problem graph, however there is no data stored inside the Pheromone Object which relates this Object to a specific edge. To enable logical indexing of these Objects such that the author could continually the pheromone data for a specific edge is a fairly trivial task however, the method of doing so has changed since the initial proposed design as seen in section 2.7.2, appendix B.

The concept of a using two-dimensional array to store the pheromone data is still present however, the dimensions of such structure are now directly representational of the number of cities present in the current problem. Assume there are x number of cities in the current problem then the length of each array would also become x. Therefore the instantiation of the pheromone data structure is of the format $Pheromone[][]pheromone = newPheromone[x][x]$. Assuming these x cities have the $index$ values $0, 1, ..., x-1, x$ then indexing any edge would be done using the format $pheromone[x][y]$. Given this format, x and y are any valid $Cityindexes$. For example, to access the pheromone concentration for the edge corresponding to the path from $City$ 0 to $City$ 5 would be accessed using $pheromone[0][5]$. As accessing an element in in an array is a $O(1)$ operation this method of access is extremely efficient and supports a problem of almost any size as the data structure resizes with the problem. Figure C.1, appendix C represents how the pheromone matrix is initialised using the initial pheromone value for all edges.

5.2.1.2 Distance Matrices

Initially there was no proposed design for the inclusion of data structures representing the distances between each node in the graph as the initial design for the problem representation made it simple to calculate the distance between the current node and the destination node extremely trivial. The author decided that as the problem representation has changed to the TSP there needs to be a better method of accessing the distances between two nodes (Cities).

The design for the distance matrices are based on the implementation provided by Thomas Jungblut [22]. The implementation of these data structures is identical to that discussed in section 5.2.1.1 however, the data stored in each array element represents the $euclideanDistance$ between two $cities$ rather than the pheromone concentration. As defined in section 5.2.1.1 the size of distance matrices will represent the number of cities. If there are x cities in the current problem then the format for instantiation will be $double[][]distanceMatrix = newdouble[x][x]$. The way in which elements are accessed also remains the same as above, however once instantiated these values will remain constant as a $City$ cannot move location. These matrices are easily populated after the cities have been instantiated. The code in appendix C figure C.2 represents how this has been implemented. A correct implementation would return a distance of 0.0 if $index[x][x]$ were to be accessed where x is the same value. This is because the distance from any $City$ to itself is 0. As the probability is represented as $p_{xy}^{k} = \frac{(\tau_{xy}^{\alpha})(\eta_{xy}^{\beta})}{\sum(\tau_{xy}^{\alpha})(\eta_{xy}^{\beta})}$ where η_{xy} represents $\frac{1}{distanceMatrix[x][y]}$ (inverted distance). As this function is used extremely often during the algorithms execution rather than converting the value of $distanceMatrix[x][y]$ every time an inverted value is needed it is far more efficient to store an inverted version of the $distanceMatrix$. The code snippet in figure C.3 demonstrates how this can be implemented. The $invertDouble(distanceMatrix[i][j])$ in said figure simply returns the value of $\frac{1}{distanceMatrix[i][j]}$ however if $distanceMatrix[i][j] \;==\; 0.0$ then 0.0 is returned as $\frac{0.0}{distanceMatrix[i][j]}$ is illegal.

5.2.2 Agent and City Collections

Both Agent and City Objects are designed to be stored using data structures which implement the List<type> Java interface. As the number of Cities and Agents present in the current problem is dynamic and user defined the size of the data structure must be established at runtime and therefore the data structure chosen must scale well and maintain performance regardless of its size. In addition to this, a user is able to modify the amount of Agents and Cties once created, therefore the data structure must be able to be effectively resized dynamically without complication. The ArrayList data structure has been chose to store the Agents and Cities as it is scalable and can be dynamically resized. An Array implementation could be used here however, if the Array reached capacity then there would need to be conditionals put in place by the author to copy the contents of the full Array into a new, larger Array instance. The use of ArrayList means that the author does not have to worry about these conditionals are they are handled for the author by the ArrayList implementation. In addition to this, the order the elements in generally unimportant, accessing a random index however is. Using an ArrayList provides $O(1)$ random element access using the provided $get(index)$ method. Also inserting into an ArrayList is $O(1)$ where as an Array implementation would provide $O(n)$ insertion as the next free index would have to be located. As the ordering in unimportant there is no reason to use a LinkedList data structure and the improved performance of the ArrayList versus an Array has enabled the author to confidently select an ArrayList implementation as the best suited data structure.

Unless explicitly stated by loading a configuration from a file, the locations of the $City$ elements in collection of cities are randomised. There are conditionals in place to ensure that two cities cannot be placed in the same location as this is a possibility if the locations of such cities is random. The implementation of the $City$ instantiation can be found in appendix C, figure C.4.

The starting location of the Ant Objects represented as elements in the collection of agents is also randomised. This random value is however limited to a range of integers with a maximum possible value equal to $cities.size() - 1$. As indexes start at 0, this enables the random function to return any $City$ index such that it is possible for an agent to start at any $City$ location. The implementation of this agent instantiation is defined in appendix C, figure C.5.

5.2.3 Best Route representation

Storing the current best route is slightly more complicated than storing the collection of Agents or Cities as the ordering is important. As the ordering is important the accessing of random indexes is irrelevant as the elements stored in this data structure when parsed in order represent the order to cities which the agent visited in order to produce this best route. The indexing of random elements would enable the elements to be parsed in an incorrect order, thus this functionality will not be used by the author and therefore the data structures performance in executing such task is completely nullified. The data structure used for this representation does not need to have dynamic resizing capabilities as the length of any best route will represent the number of cities in the problem as an agent must visit every City once and only once. The author reduced his choice to using an ArrayList or a LinkedList implementation. A LinkedList implementation has massive performance enhancements when compated to an ArrayList as a LinkedList can provide constant time ($O(1)$) insertion and deletion operations using iterators which can be relied upon regardless of the magnitude of such data structure in combination with the lack of regard to element ordering. An ArrayList cannot reliably provide such performance. This in addition to the fact that the ordering of elements is important, has lead the author to decide on using a LinkedList data structure for the representation of the best routes.

5.3 Pheromone Operations

The translation of the pseudo-code described in algorithm 7 was a fairly simple process as the pheromone algorithm is itself relatively simple. Once implemented the author quickly realised that the pheromone operations (deposit and decay) were not behaving correctly. During the course of agents $move()$ method every time the agent moves from its current location to its next location pheromone proportional the distance travelled would be deposited on the corresponding edge. The problem was that every time this pheromone was deposited the global pheromone was in fact being decayed so the global pheromone was being decayed every time the agent moved. In order to prevent this problem from arising rather than directly updating the pheromone on the edge when the ant moves the agents pheromone deposit on this edge is instead added to a secondary $newPhero$ variable in the Pheromone Object for this corresponding edge. This variable is used to collate the total amount of new pheromone to be deposited for any edge. When it is time to update the edge pheromone, every Pheromone Object is iterated through and value present in this $newPhero$ variable is used in the pheromone calculation allowing for correct behaviours to be represented. This is slightly more complicated than the author envisioned however, the additional complexity is essential to ensure correct behaviour. The implementation of this method is shown in appendix C, figure C.6.

5.4 Solving the Problem

The author encountered unforeseen complications when implementing the pseudo-code represented in algorithm 1. The subsections below described the problems with the implementation process and their associated solutions.

5.4.1 Agent Movement

The author has problems initially with his implementation of the probability based movement of the agents. The implementation of the Ant's $move()$ function worked as expected there were no complications associated with that however, the complications that the author encountered involved the probabilistic selection of the Ant's next location. The problem caused every Ant that started at the same City index to take exactly the same route. Upon debugging the Ants $getNextProbableNode(int\ y)$ function the author realised that there was in fact no random element associated with this method and the City with the highest probability was constantly selected thus, the Ants never had the ability to divert away from the best looking route causing local solutions to be present. The author attempted to implement this random element into his method however the algorithm did not perform as efficiently as expected. Thomas Jungblut had provided an open source implementation of a basic Ant Colony Optimisation algorithm [22]. This implementation was provided free of charge and allowed modification which enabled the author to replace his lacklustre node selection function with the one provided in this working implementation. This node selection function did however require modification by the author to ensure correct operation with his architecture. Once this method had been modified and implemented the node selection process for the Ant's next location worked as initially intended. The author realised that his implementation took completely the wrong approach which involved sorted the candidates locations based on their associated probability. Had the author not accepted that the re-use of a working implementation is the best way to progress the application then a reduction in project quality would have been a possibility. The implementation of this node selection method can be seen in appendix C, figure C.7. If this method returns -1 then the Ant is deemed to have completed its walk.

5.4.2 Automated Solving

Implementing an algorithm which once started, executes until completion was in fact fairly simple to do initially. This simplistic implementation stemmed from the fact that initially the algorithm only considered the fact there was one iteration. As there was one interaction it was simple to define a stop condition which was in fact when every agent had completed its walk which meant the algorithm stopped when $Ant.getFinished()\ returned\ true$ for every Ant. Once the author had implemented automatic execution for one interaction, the ability for a user defined number of interactions was implemented. The addition of to support x number of iterations where $x > 1$ introduced several new complications. Each new iteration meant that the Ants need to be reset so that they forget everything from the previous iteration(s) but the pheromone levels, Cities and current best route must all be persevered across iterations. In order for this to be possible, the author adapted to algorithms stop condition to now be relative to the number of iterations or in other words stop when $currentIteration == totalIterations$. A new iteration is started when all Ant's have finished their walk, instead of checking every Ant for the condition $Ant.getFinished() == true$ once an ant had finished a variable representing the total number of Ants current working (unfinished) was reduced by 1. When this value reaches 0, there are no Ants currently working thus the iteration has completed. Once an iteration has completed the Ants must be reset so the next iteration can start, this is done by calling the $initAnts()$ method again on the current $World$ enabling the old Ants to effectively be reset. The variable representing the number of Ants working is also reset so that $antsWorking == totalAnts$ and the process starts again. During this reset operation the pheromone levels, cities and best route remained unchanged to ensure previous iteration results remain. The implementation for this algorithm can found in appendix C, figure C.8.

5.4.3 Step Based Iteration

In order to implement an algorithm which executes on a step-by-step basic will require constant user interaction in order to execute to completion. In order to achieve this, several modifications have been made to the algorithms used in the process of automated solving. The first problem associated with step based iteration is the fact that an Ant must be stopped moving after it has moved to its next location, in comparison the fully automated approach uses an algorithm which ensures an Ant continues moving until it is finished or move while the algorithm in figure C.7 does not return -1. In order to stop the Ant after one movement the algorithm in section 5.4.1 can remain largely the same, however this while loops must be removed so that the movement function executes once and only once per step. This change alone however is not enough to stop the algorithm from executing until completion. Rather than using a $for\ each$ loop to iterate through every Ant and instructing it to move as described in figure C.8 there must now be a way to track the current Ant which is moving and update this tracker once said Ant is finished. In order to do this an index representing the location of current working Ant is stored so that when the algorithm is stepped through only the Ant and that index is signalled to move 1 step. Once this Ant has finished, then this index is incremented until $index >= agentsCollection.size()$. Similar to the algorithm described in section 5.4.2 once every Ant has finished walking, the iteration is complete and if there are more iterations to execute then the Ants and index are reset. Although this feature wasnt part of the initial or improved design, the ability to re-use the existing algorithm enabled the implementation of the feature to be far less complex. The implementation of this step based iteration can be seen in figure C.9.

5.5 Rendering the Algorithm

The implementation of the pseudo-code represented in algorithm 4 turned out to be far more complicated than the author actually planned. The subsections below segment the complications that arose and the steps the author took in order to ensure a proper visualisation process.

5.5.1 Swing Worker

Initially the algorithm was executed on the AWT event-dispatching thread. The automated solving algorithm is a fairly long and somewhat resource intensive process and given that the graphical user interface elements are also executed on this thread. This causes problems during execution. During the algorithms execution time as the interface and algorithm are both on the same thread, the user interface would freeze and become unresponsive until the algorithm had completed. The implication of this are that literally none of the algorithms execution is visualised, which is the complete opposite of the intended implementation. The author realised that this is extremely bad design and in order to correct such complications concurrency should be introduced into the system. The graphical user interface is built up of several Swing component and these components are not thread safe which can cause some concurrency problems. As a result of these components not being thread safe the Java language is defined such that any access to these components must be done from within the same thread which happens to be the Event Dispatch Thread. In order to both modify the user interface and execute the algorithm in a concurrent manner the author has used the $SwingWorker$ Object provided by the Java language specification. A $SwingWorker$ is designed to perform length tasks which interact with a graphical user interface adhering to the language specification protocols. The $SwingWorker$ has a $doInBackground()$ method which executes this method inside a separate $Worker$ thread. During the execution of the $doInBackground()$ method any interactions with the Swing components are dispatched on the correct Event Dispatch Thread by the $SwingWorker$. As a result the author has been able to implement concurrency to allow for the parallel execution of the algorithm and interface modification relatively complication free. The code contained within this applications $SwingWorker$ is shown in figure C.8 and is the location of the algorithm discussed in section 5.4.2.

The step-based iteration method of solving does not need to worry about such complications as each step takes very little time to complete, thus the addition of a $SwingWorker$ to facilitate such behaviour is over complicating the situation and will not benefit the user in any way.

5.5.2 Visualising Agent Movement

One of the major problems associated with the visualisation of the algorithms execution is accurately representing the agents movement between two cities. Initially the agents would move between cities without any indication to the user about which path the agent took as the agents themselves would seem to teleport from one city to the next. This is not the kind of feature that should be present in an application of this kind. The author decided that there needed to be a way to visualise the Ants movement between cities however, the visualisation of this movement should be simple so that it doesnt over complicate a simple action such as moving from one point to another. Given that every Ant stores the indexes of the last two cities it visited in a variable $movementTracker$. This variable is an array of integer values of size 2 and it will only ever contain the last 2 cities the agent visited, thus $movementTracker[0]$ represents where the Ant started prior to moving and $movementTracker[1]$ is where the Ant finished.

The author decided that as the coordinates of the two cities stored in this $movementTracker$ variable are fairly simple to obtain it would be straight forward to use linear interpolation to draw a visual aid along the Ant's movement path in order to highlight which path the Ant took. Given a start and end coordinates for a line Linear Interpolation can be used to find the coordinates for any point along that line.

$$result_x \;=\; x_1 \; X \; (1 \; - \; \mu) \; + \; x_2 \; X \; \mu$$
$$result_y \;=\; y_1 \, X \, (1 \, - \, \mu) \, + \, y_2 \, X \, \mu$$

Figure 5.2: Formula for Linear Interpolation where x_1, y_1 are the start coordinates, x_2, y_2 are the end coordinates and μ is the location to find [17].

Once the coordinates of the two cities stored by index in the $movementTracker$ variable have been retrieved they can be passed to the $LinearInterpolateX(double\; x1, double\; x2, double\; mu)$ and $LinearInterpolateY(double\; y1, double\; y2, double\; mu)$ methods in order to find a point on the line between these two City locations, this line is representive of the path the Ant will take to travel between said locations. The correct μ value must be selected in order to provide the best representation. The author opted to go for drawing a series highly visible ellipsis along the line with the distance between said ellipsis to be 0.2 which is the equivalent to approximately 20% of the lines total length. These ellipsis will therefore be drawn using rhe retuned values from the equantions in figure 5.2 where $\mu\; = 0.2,\; 0.4,\; 0.6,\; 0.8$. This will enable the user to clearly identify which path has been taken and also enable the user to judge the length of said path as the longer the path the greater the spacing between the ellipsis. The code implementation of the equations in figure 5.2 is shown in figure C.10 and the code for drawing the path the Ant takes is shown in figure C.11.

5.5.3 Scaling

In order for the visual representation to be rendered in a suitable manner there has to be some form of scaling to ensure all the visual elements remain in aspect to each other. The size of the canvas area is fairly limited so the author had to decide on a scale factor which enabled the complete representation to be container within the canvas bounds. The author decided that as the canvas area is $800px$ in width and $600px$ in height a scale factor of 20 is perfectly suited to this environment. As 20 is a factor of both 800 and 600, this will effectively provide the author with a grid for 20 x 20 squares which is 40 x 30 in dimension. If a scale factor greater or less than 20 were to be used then the dimensions of this grid would change and if this scale factor was too large there would not be enough room to display an adequate amount of detail. There are also complications with using a scale factor which is too small, this will cause the visual elements to be difficult to see as they will appear very small on the canvas which may alienate some users.

This scale factor will be applied when rendering any element on the canvas which includes the rendering of City locations, Agent locations and drawing paths between cities. If a City has the (x, y) coordinates of $(18, 15)$ and the author used no scale factor to scale this when rendering then the City would be rendered at $(18px, 15px)$. If a further two cities had the (x, y) coordinates of $(30, 5)$ and $(2, 9)$ respectively with no scaling applied these cities would be rendered extremely close to each other which would make it difficult to visualise the algorithm execution as the world would occupy an extremely small amount of canvas real estate. Applying this scaling factor to same set of cities and coordinates causes the cities to be rendered at $(360px,\; 300px)$, $(600px,\; 100px)$ and $(40px,\; 180px)$. This produces a world where the cities are suitably spaced such that there is sufficient room to render the algorithms execution and visualise the Agents movements between such cities. This scaling factor is hidden from the users. As far as the user is concerned when creating a file for use in loading a problem configuration a cities (x, y) location is specified in terms of its grid square location rather than in terms of scaled coordinates as shown in section **??**.

5.5.4 Painting the Canvas

Every aspect discussed in the section is vital in the painting of the algorithms execution for the reasons stated above. As every aspect of the algorithm needs to be visually represented the actual painting of the algorithm to the canvas became more complex than the author initially envisioned.

Representing the City locations cities is fairly simple to implement as every City stores the location for its (x, y) coordinates. In order to render each City at its correct location, the collection of cities is iterated through and the method $drawOval(City.XX20,\ City.YX20, 20, 20)$ is used to draw an oval which has the width and height of $20px$ and is represented in the correctly scaled location. There is a slight problem with this implementation however. As discussed in section 5.5.3 the canvas is segmented in a grid 40 x 30 in dimension therefore the coordinates representing a grid cell represent the top right corner of such cell. If a City is drawn at $(20, 10)$ then this will be drawn in the top right corner of the grid square $(20, 10)$. The author found this to be visually off putting and didnt produce the look the author wanted. In order to prevent this the location of the drawn city must be offset such that rather than being drawn in the top left of grid location it is then drawn in the centre of such grid location. In order to do this a simple change to the $drawOval$ is methods parameters is required. The modified version of this method call now resembles $drawOval((City.X * 20) - 10,\ (City.Y * 20) - 10, 20, 20)$ where 10 represents $\frac{scaleFactor}{2}$ this will now draw the cities centred into the correct grid square. These cordinates are also used to render the ants at a city location. As every Ant stores the index of the City it is current at, it becomes trivial to found the locaton where it should be drawn by iterating through the collection of cities untill the index is found.

Representing the path between these cities is also a somewhat simple task. These routes are used to visually display the routes which any agent can take to navigate between any two cities. As the graph is being represented as a fully connect graph where any City can be reached from any other City, the paths displayed to the user must reflect this. Each path will be displayed as defined in section 5.5.2 which is as a line drawn from one City to another. The collection of cities will be iterated through, and for each City in the collection the collection of cities will be iterated through again. The first iteration of the collection will represent the start location for each line (path) then whilst keeping this location the same, the next iteration of the collection will be used as the end location for the line (path). This is a simple way to draw a line from each City to every other City.

As these lines represent the paths between cities, they are therefore used to represent the edges connecting nodes (cities) in the graph and as a result these lines are also to be utilised for the visualising of the pheromone concentrations on such edges. In order to represent the pheromone the author has decided that the opacity will change in proportion to the pheromone concentration on their given edge. As the pheromone values for a given edge can become incredibly small $(<< 0)$ depending on the number of iterations, cities and agents the opacity of such lines must be based on a suitably scaled version of the pheromone values on each edge. The author decided use a scale factor of 2000 when applying the pheromone concentration to the opacity of the corresponding line however, the opacity (alpha) level for a given line is an 8-bit integer thus 255 is the maximum value. To prevent conversions of $pheromoneConcentration_{xy}\ X\ 2000\ >\ 255$ causing problems there is a condition to catch larger values and reduce their size to maximum legal value of

255. As the pheromone concentration gets lower and lower the paths with start to become invisible to the user so they can easily see which paths are actually being used.

Displaying the best route to the user is the most complex part of the visualisation process and also one of the most important. The implementation of this feature was far more complex that the author expected and its complexity extends further than simply drawing a red line. As a line requires a start and end location this means that the best route data structure (see section 5.2.3 must be iterated in pairs, and accessing these pairs is not as elegant as the author has planned. The best route will be created using a series of lines drawn between such pairs. Once the first city of such pair has been identified the next element in the data structure is the second City belonging to the pair however, the author first has to check if there is infact a next element before such an action can take place which is simply doing using a condition similar to $(elementIndex + 1) < bestRoute.size()$. If this condition returns true then continue with the line drawing process as there is a valid pair of cities to draw a line between. Once this pair of cities have been identified it is simple to draw a line between them as the coordinates are known. The complexity comes from the multiple iterations of the City collection in order to locate the City pairing, this makes the code block slightly more difficulty to both debug and mentally visualise which made this feature slightly harder to implement. The code for such a feature is shown in figure C.12.

5.6 Elitist Ants

The addition of Elitist Ants was slightly more complex than the author initially thought however, as a whole it is not an overly complex feature. The algorithm remains largely unchanged from the Basic Ant System, the added complexity comes from the fact that pheromone needs to be deposited on the Elite Ant best routes every iteration. Correctly storing and updating the current x number of Elite Ants was simple to implement for the author and involved checking to see if this agent should be an Elite Ant or based on its total distance travelled. The implementation for this is shown in algorithm 2, lines 19 to 24 as remains as designed. As discussed with the rendering of the best route, the deposit of pheromone along can be done so using a pair based iteration. This is slightly less complex that the rendering process as there is no need to retrieve the City coordinates, As the pheromone matrix is set up in a manner such as that stated in section 5.2.1.1 only the City indexes are needed. Once a pair of City indexes has been extracted from the best route pheromone can be deposited along the route in a fashion similar to this $pheromone[CityPair1.index][CityPair2.index].addToNewPheromone(pheromone$ $[CityPair1.index][CityPair2.index].getPheromone + e)$ where e is a constant representing $\# \; of- \; cities$. The code for depositing pheromone of the x number of elite routes is shown in figure C.13.

5.7 Uphill Path Generation

The implementation of uphill paths was itself fairly simplistic and involves the process of locating the route's corresponding element in the distanceMatrix and doubling the value contained in such element. This enables an uphill path to effectively cost double its normal cost. Depending on the current β value, the agents will generally have a higher probability of taking a shorter path. As this is the case the addition of such uphill paths allows the algorithm to potentially produce a wider range of solutions to the same problem configuration.

The selection of these uphill routes is slightly more complex than the author assumed them to be. The location of these uphill paths is randomly generated. The location is determined by using the $Random.nextInt(\#\ of\ cities)$ method to randomly select an element from the distance matrix. This element represents an edge length between two cities, thus doubling this value will effectively double the distance between said cities making this edge cost more to traverse than usual. There are problems with using a random selection process for selecting the start and end locations for this uphill path however. These start and end locations will be valid city indexes which means that these two randomly generated values cannot be the same as there is no edge from a City to itself. This is where the added complexity is introduced. The author must now check that the randomly generated values are in fact different, if they are equal then the random generation process must happen again. As the random value generated is in the range of $0...cities.size()$ how likelihood of these random values being the same is dependent on the magnitude of the value range. If there are few cities then this random generation process make have to be executed an inefficient number of times until two non-identical values are generated, this is not really a problem if there are a large number of cities. The code representation for the generation of uphill routes is shown in figure C.14.

Chapter 6

Testing

6.1 Overall Approach to Testing

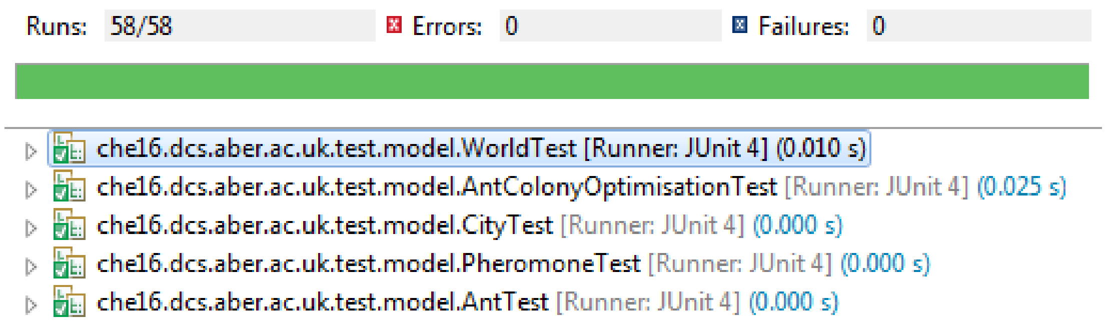
This chapter presents a multitude of different testing types. There are authomated unit tests to ensure the internal code logic behaves as expected. The User Interface tests are tests which are difficutly to automate but are used to esnure that the user interface encaptures the correct state of execition. Acceptance testing is used to assess how well the applications meets the specified requirements in section 2.3. Stress testing has been considered by the author but ultimately has been omitted. The author feels that the application is never subject to heavy loads, nor can the user exceed the normal operational load for the application thus, there is no need to stress test currently.

6.2 Unit Tests

Unit tests were developed using the JUnit framework and have been designed by the author to test a series of boundary conditions and any underlying algorithmic logic present with the application code. The execution of these tests than then be fully automated and using the built in interface provided by Eclipse a summary of the completed tests can be viewed by the author enabling clear identification of failed tests and reason for failure. The unit tests are generally only performed on the Model package. The view package is difficult to unit test as it requires user input and is therefore manually tested by the author. The controller package does not really contain any behaviour worthy of testing as it mainly consists of simple assignments or the calling of other system functions.

There are features present in the application code which the author has excluded from the unit testing process. Any feature which relies on random numbers to perform its task has not been unit tested. As a random number is used the author feels that they cant reliably be tested, therefore it doesnt makes sense to attempt to. Instead the author has extracted the code which relies on such random numbers and tested the behaviour of such code using specifically defined values, this enables the author to accurately test for expected behaviours. In addition to this getter and setter methods or simple assignment operations have not been tested. The reason for excluded these methods from unit tests is because the code responsible is so simple it cant fail so there is no real purpose in producing such tests.

The unit tests that the author has produced revolve around ensuring that only legal algorithm parameters are accepted and that the movement, probability and pheromone functions are behaving as expected and return legal values. These tests contain validation checks against a range of values for each parameter including boundary conditions to ensure that they are correctly dealt with, examples of such testing can be seen in figures D.1 to D.3. The probability and pheromone functions are tested to ensure the correct value is returned and in the case of probability, ensure that the sum of all calculated probabilities is equal to 1, such a test is shown in figure D.4. There are many more tests present in the test package for the application however the author has only briefly outlined a small number of these. Evidence of such testing can be seen below in figure 6.1.



```
Runs: 58/58        Errors: 0        Failures: 0

▷ che16.dcs.aber.ac.uk.test.model.WorldTest [Runner: JUnit 4] (0.010 s)
▷ che16.dcs.aber.ac.uk.test.model.AntColonyOptimisationTest [Runner: JUnit 4] (0.025 s)
▷ che16.dcs.aber.ac.uk.test.model.CityTest [Runner: JUnit 4] (0.000 s)
▷ che16.dcs.aber.ac.uk.test.model.PheromoneTest [Runner: JUnit 4] (0.000 s)
▷ che16.dcs.aber.ac.uk.test.model.AntTest [Runner: JUnit 4] (0.000 s)
```

Figure 6.1: Evidence of the completed execution of the applications unit tests. The view present is the built in Eclipse JUnit interface.

6.3 User Interface Testing

It is difficult to reliably test the graphical user interface using a white-boxing testing approach such as the automated unit tests process as mentioned above. To overcome this obstacle the author has decided to manually test the user interface using a black-box testing approach to ensure the correct behaviour to exhibited. The author feels that for the scope and time frame for this project, manually testing the display is the most appropriate form of testing.

During this test process the author did not repeat any form of test that had been previous executed in the unit testing process as these tests are already proved to pass. The tests performed manually by the author will be for the purpose of ensuring that the correct error messages are displayed to the user in the correct situation and making sure the algorithm is being correctly rendered and its current state is modified in as close to real time as possible.

In addition to testing for errors the file IO process will be tested ensure that configurations can be correctly saved to a specified file and if a user specifies a file to load a configuration from, this file can in fact be handled even if the contents of such file contains incorrect values or bad formatting. Several test files will be produced, each file will be designed to test a variety of possible illegal file contents. Tests will be done to ensure that the application rejects files that contain; missing data, incorrect type for a specified value and City coordinates which exceeds the bounds of the canvas. Files containing such malformed data should not cause the application to crash this will be tested. The saving of a configuration to a file doesnt not need to be as rigorously tested as the algorithm has to be instantiated with legal values before the writing process can be executed therefore, the values of the written file will be legal and of the correct format. An example of such file contents can be found in section 4.2.1.

The full list of black-box test performed by the author can be seen in section 4.2.2 which includes testing against the function requirements. The detail and coverage of such test could be extended however the author feels that for the current state of this project this will suffice.

6.4 Stress Testing

6.5 Acceptance

Detailed descriptions of every test case are definitely not what is required here. What is important is to show that you adopted a sensible strategy that was, in principle, capable of testing the system adequately even if you did not have the time to test the system fully.

Have you tested your system on real users? For example, if your system is supposed to solve a problem for a business, then it would be appropriate to present your approach to involve the users in the testing process and to record the results that you obtained. Depending on the level of detail, it is likely that you would put any detailed results in an appendix.

The following sections indicate some areas you might include. Other sections may be more appropriate to your project.

Chapter 7

Future Work

Chapter 8

Evaluation

Examiners expect to find in your dissertation a section addressing such questions as:

- Were the requirements correctly identified?
- Were the design decisions correct?
- Could a more suitable set of tools have been chosen?
- How well did the software meet the needs of those who were expecting to use it?
- How well were any other project aims achieved?
- If you were starting again, what would you do differently?

Such material is regarded as an important part of the dissertation; it should demonstrate that you are capable not only of carrying out a piece of work but also of thinking critically about how you did it and how you might have done it better. This is seen as an important part of an honours degree.

There will be good things and room for improvement with any project. As you write this section, identify and discuss the parts of the work that went well and also consider ways in which the work could be improved.

Review the discussion on the Evaluation section from the lectures. A recording is available on Blackboard.

Appendices

Appendix A

Requirements Specification

1.1 Introduction

1.1.1 Purpose

The purpose of this document is to give a detailed description of the ‘Visualising Ant Colony Optimisation’ application. This document will cover the interface interactions and methods as well as providing definitions to important terminology. The document is primaraly intended to be used as a reference point for the initial stages of development.

1.1.2 Scope

The ‘Visualising Ant Colony Optimisation’ application is a desktop application which is designed to demonstrate the behaviour of an underlying ACO algorithm given various algorithm parameters. These parameters will be defined by the user and can be modified at their convenience. The algorithms behaviour given these parameters will be visible and the users can make a clear assessment about how each of the parameters impacts the algorithms performance.

During the background research into this subject area there does not seem to be too many applications which offer ACO visualisations and there are even less which provide a ‘friendly’ environment which is simple and intuitive to use regardless of the users background knowledge in regards to the subject area.

The software will be deployed in an educational environment, and aims to provide a means for teaching ACO to students as part of an Artificial Intelligence course or for independent use as a self-learning exercise. As a result the software must cater for the majority of user groups to maximise its effectiveness. This means the software must be accessible on all major platforms and perform equally well on said platforms.

Term	Definition
ACO	Ant Colony Optimisation
user	Anybody who is using or wishes to use the software
user group	A collective group of users representing different user needs.
interface	A graphical user interface
standalone	Operates independently of other hardware or software
agent	The entities which will be traversing the graph

Table A.1: Definitions for the keys terms used throughout this document

1.1.3 Definitions

1.2 Overview

This section will give an overview of the proposed application. This section will also expand on the expected user groups and functionality required by said groups. The constraints

1.2.1 Product Descriptive

The software application will be standalone and does not need to communicate with another system or application, because of this there is no need for any form of network connection to be present in order to use the application to its maximum potential.

The application will communicate with the Operating System on the host machine in order to enable the save and load functionalities through simple file input and output. However the users access to the host machines file system will be restricted by the fact that the saving and loading will be restricted to the users home directory preventing the overwriting of important documents.

The application itself will not take up too many system resources even if a large problem is being handled. This allows the users on a system or network to run the application without it impacting the performance of other services. Given that the target audience is educational establishments this is especially important as many teaching fellows have multiple applications running during a lesson or lecture and a negative impact on their system could reduce the amount of information taught during said session.

1.2.2 Product functionality

The users will be able to view a world which represents the Agents and the nodes in the graph otherwise known as the world. The state of this world will be directly related to the algorithm parameters specified by the users using the interface provided. There are several parameters which can be modified by the user, each of which will have a different impact of the state of the word and the algorithms behaviour.

The parameters which can be modified will be clearly labelled and will be obviously editable. As these parameters will be user defined there will be strict error checking measures in place to catch any illegal values before they can cause problems for the algorithm, and in addition each parameter will have a range of legal limits applied to them. This will prevent the users from entering values of the incorrect type (String when the systems needs a double) and will also prevent values from outside of the specified range being accepted. When a complication or error arises there will be a simple error message presented to the user informing them of both the error and why it occurred which should enable the user to resolve what they did incorrectly.

1.2.3 User Groups and Characteristics

There will be three main user groups associated with this application. Each of these user groups will interact with the application in a different manner but the main purpose and result will remain the same.

Teachers/Teaching fellows will use the application with the underlying knowledge already in hand. They will be mainly be using the applications to visually portray ideas and will have expectations in regards to what to expect for a solution and will have some idea how the parameters impact the final result.

Students will use the application with some background knowledge of the underlying concepts but will still use the application in an experimental manner and may have little expectations or understanding of how changing certain parameters impacts the final result.

People new to the subject area will use the application with potentially no idea about the underlying concepts. There will be measures in place to explain the underlying metrics and give an insight into what the application is actually doing. Given that they have less knowledge of this subject area that the other two user groups mentioned above, they will still be able to achieve the same results and levels of functionality. The application will cater for all users regardless of prior knowledge.

1.2.4 Constraints

As the application is standalone is reduces the amount of constraints which it becomes subject to. The main constraint which the application is associated with is the dimensions of the users display. The interface has to house a lot of elements in order to produce a simple and effective environment, thus it takes up quite a lot of screen real estate. However in modern times the amount of space required for the application to perform as expected is far from unreasonable and the application will be developed with this in mind.

The algorithms execution time directly proportional to the user defined parameters, more specifically the number of agents and the number of nodes in the graph. The more agents and nodes the larger the execution time and resource requirements will be. The application will be developed with this in mind as there will be a constraint on how much memory and system resources the application should use. There is no expectation on the user to have a superfast high end machine therefore the application will be designed to accommodate a standard machine for these modern times and correct limits will be placed on these user parameters.

The application does write the host systems file store so there must be adequate room to do so, however the files that will be written are simple text files which will not take up a lot of room on the host machines disk. Depending on the users machine this could still be a constraint. The responsibility and handling for this will belong to the users machine.

1.2.5 Assumptions

It will be assumed that the user will have the correct drivers installed and their machine will be able to handle the algorithms execution. The applications algorithms are not too resource intensive, therefore this is a reasonable assumption given the modern era and the advancement of computer technology.

It will be assumed that users will meet the minimum display requirements thus no dynamic resizing of the interface based on the users display dimensions will be performed. This significantly reduces complexity.

Another assumption is that every user will have some experience of using similar software and the interface will be familiar and therefore will be easy to use and navigate. The interface will use traditional methods such as simple buttons, text boxes and drop down menus to provide the user access to certain functionality.

1.3 Specific Requirements

This section contains the function requirements of the software as well as giving details about the different interfaces.

1.3.1 User Interface

There will be one main user interface for this application. This interface will contain a display area which is where the algorithms current state of execution will be represented visually to the user. There will also be an area which will house the text boxes which will be used by the user to interact with the algorithm and modify the parameters.

The display area will be simple and will be clearly identifiable. The text boxes responsible for handling user inputs will be clearly labelled so the user knows exactly what parameter they will be modifing. The text boxes will be obviously editable and the user will associate the look and feel of these text boxes with the fact that their contents can be modified.

The display will represent everything about the algorithms current state. This will display all of the graphs Nodes and all of the Agents at their current Node. The display area will also show the current pheromone levels for each connecting edge for a given node, this will be done in a way that is clear and understandable by all of the user groups mentioned in section 1.2.3.

There will also be a textual representation of the current best agent. This will display the best route and the distance of the best route and will update as the global best is updated. There will also be textual representations of how many agents are currently working in addition to how many agents have finished.

1.3.2 Hardware Interface

The application does not require any specific hardware or host environment as it will be fully cross-platform compliant there are no direct hardware interfaces. There is no network use in this application thus there is no need to communicate with network adapters or anything of similar nature. The system interactions between this application and the host's Operating System file system will be delegated to the Operating System itself.

1.3.3 Functional Requirements

ID: FR1
Title: Launch the Application
Description: Regardless of the users host environment the application should be able to be launched by the user using an executable .jar file.
Dependencies: None

ID: FR2
Title: Generate a World
Description: The user must be able to randomly generate a world for the algorithm to be executed on.
Dependencies: FR1

ID: FR3
Title: Visualise a World
Description: The user must be able to visualise the world and its parameters including the number of agents, the agent locations and the number of Nodes.
Dependencies: FR1, FR2

ID: FR4
Title: Provide a means to modify parameters
Description: The application must provide simple ways to modify the algorithms parameters.
Dependencies: FR1

ID: FR5
Title: Generate a World with specified values
Description: The user must be able to generate a world for the algorithm to be executed on. This World with have user defined properties such as the number of Nodes and Agents.
Dependencies: FR1, FR4, F10

ID: FR6
Title: Visualise the Pheromone
Description: Every edge in the graph will have its own pheromone value. This must be visually displayed to the user and correctly model the pheromone deposit and decay operations.
Dependencies: FR1, FR2, FR3, FR13

ID: FR7
Title: Visualise the Agents movement
Description: As the algorithm is executing the Agents will move through the graph. The movement of these Agents must be displayed to the user in a logical manner.
Dependencies: FR1, FR2, FR3, FR8

ID: FR8
Title: Start the Algorithm's execution
Description: There must be a simple way for the user to start the algorithms execution.
Dependencies: FR1

ID: FR9
Title: Stop the Algorithm's execution
Description: There must be a simple way for the user to stop the algorithms execution anytime the user wishes to.
Dependencies: FR1

ID: FR10
Title: Validate parameters
Description: As the users will be able to define thier own parameter values there must be correct measures in place to ensure the values entered are legal. If they are indeed illegal then suitable error messages will be displayed.
Dependencies: FR1

ID: FR11
Title: Display the best path
Description: As the algorithm is performing its task there must be a way to display the current best result to the user.
Dependencies: FR1, FR2, FR3, FR5, FR7, FR8, FR12

ID: FR12
Title: Agents must be able to move between Nodes
Description: In order for algorithm to perform as expected there must be a way for each Agent to move between Nodes in the graph.
Dependencies: FR1, FR2, FR3

ID: FR13
Title: Model Pheromone operations
Description: There must exist a way for the algorithm to correctly deposit and decay pheromones on graph edges adhering to specific formulae.
Dependencies: FR1, FR2, FR3, FR11

ID: FR14
Title: Load Configuration from a file
Description: There must exist a way for the users to load a pre-existing configuration from a file

of their choosing. This allows the algorithm to be executed on the same problem multiple times.
Dependencies: FR1, FR2, FR3

ID: FR15
Title: Save Configuration to a file
Description: There must exist a way for the users to save the current configuration to a file of their choosing. This allows the algorithm to be executed on the same problem multiple times.
Dependencies: FR1, FR2, FR3

ID: FR16
Title: Exit the application
Description: The user must be able to exit the application completely killing the process and freeing system resources.
Dependencies: FR1

1.3.4 Requirement Evaluation

Each of the functional requirements mentioned in section 1.3.3 differ in thier levels of importance. The dependencies field for each requirement donates which requirements must be completed before said requirement itself can be finished. As this is the case the following represents the order of importance for each functional requirement.

Requirement	Dependant Requirements
FR1	FR2, FR3, FR4, FR5, FR6, FR7, FR8, FR9 ’ FR10, FR11, FR12, FR13, FR14, FR15, FR16
FR2	FR3, FR6, FR7, FR11, FR12, FR13, FR14, FR15
FR3	FR6, FR7, FR11, FR12, FR13, FR14, FR15.
FR8	FR7, FR11, FR13
FR4	FR5
FR5	FR11
FR10	FR5
FR12	FR11
FR13	FR6
FR7	FR11
FR11	
FR6	
FR9	
FR14	
FR15	
FR16	

Table A.2: Table representing the Functional Requirements and which other requirements are dependent on them.

As described in Table A.2 the number of dependant requirements a requirement has the more important it is to the progress of the application. FR1 is the first task that must be accomplished therefore every other requirement is dependent on this being completed. FR1 is the ability to launch the application, if the application cannot be launched then none of the other requirements can be completed, this is a critical requirement.

FR2 is another critical requirement which must be completed early in development as many other requirements are dependent on its completion. FR2 is the ability to generate a World. A World contains all the data that the algorithm needs in order to both execute and visualise, therefore if there is no way to generate a World there is no way to visualise or model the algorithms execution (FR3).

Apart from FR8 (start the algorithms execution) the remaining functional requirements are somewhat independent of each other and are less critical. However this does not mean that they can be avoided as they will be needed in the final application version.

Appendix B

Initial Design Proposal

2.1 Introduction

2.1.1 Purpose

The purpose of this document is to give a detailed description for the design of the'Visualising Ant Colony Optimisation' application. This document will cover the proposed interface designs and interaction methods, choice of language and the underlying data structures and logical modules used in the application.

2.2 Language

The choice of implementation language for both the graphical user interface and the Ant Colony Optimisation algorithm is extremely important. The nature of the project suggests that an Object Orientated approach would best suit as the implementation language. One of the main reasons for this is because the manipulation and use of Objects in such languages allows for Object-based decomposition allowing for more logical modules system wide when compared to a non-Object Orientated approach. A non-Object Orientated approach may be more subject to functional decomposition (the application is split into modules grouping similar functions rather than representing separate Objects).

An Object Orientated approach has been identified as most suitable. Therefore there are two main languages which are at the forefront of the selection process. These two languages are C# (*C Sharp*) and Java. Both languages have the potential to achieve a high level of success when applied to the projects problem however; they both have different consequences depending on the environment and application in which they are used. C++ (*C plus plus*) is another popular Object-Orientated language. C++ has been discounted due to lack of language experience therefore using a more familiar language such as the two specified above (Java and C#) is far more appropriate in this instance.

One of the major factors in deciding on the implementation language is the suitability of the languages features when applied to the projects problem including any external resources or compatible libraries. Both Java and C# are very similar at an abstract level in terms of provided features by default. Both include everything that would be necessary to implement the proposed design for this application. Both languages provide the ability for Objective decomposition and allow for polymorphic behaviours (multiple entities of different types using the same interface) which enables effective use of inheritance to allow multiple Agent variations (Ants in this case) to be supported easily.

C# has been created and is continually being developed by Microsoft and is focused around the .NET framework which is also a product of Microsoft. As C# is heavily Microsoft orientated its cross-platform capabilities are significantly reduced. The .NET framework(s) have only recently been open sourced so they lack full support on all platforms reducing the applications cross-platform reliability. The project is being developed with a focus on educational value; therefore cross-platform reliability is very important as maximisation of potential consumers is important (more people using the application implies more people are learning). A standard build will not be specified or assumed so there must be necessary measures in place to accommodate as many environments as possible. C# is heavily coupled with Windows based systems therefore; If C# were to be used there is a risk of alienating Macintosh and UNIX users. There are attempts to port .NET to other architectures (for example, Mono [15]) but the implementations of such approaches are not exact replicas of Microsoft .NET framework(s), therefore they cannot be relied upon. The use of Java would eliminate the cross-platform support issues as Java applications execute inside the Java Runtime Environment (JRE) which is available across most platforms and behaviours can be accurately modelled and predicted in the vast majority of cases.

Little differs between Java and C# in terms of feature presence (abstractly, how each language achieves each task is very different) thus, Java will be used for this project. The cross-platform constraints that come with the use of C# are not balanced by any necessary exclusive key features. As a result Java is the most appropriate language for the project, this all but ensures cross-platform reliability without sacrificing any important libraries or features.

2.3 Architecure

There have been considerations as to what key elements will be present in the composure of a suitable underlying architecture for the application. The architecture must accommodate both major elements of the application (graphical user interface and the Ant Colony Optimisation algorithm) in a manner that enables the best possible expansion/modification opportunities to accommodate any additional features or unforeseen changes. Selecting relevant Design Patterns will enable the above goals to become reality however; design patterns should be respectfully and must represent a general solution to a problem. The Overuse or misuse of such patterns can cause significant complexity issues through the system, this needs to be avoided.

2.3.1 Design and Architectural Patterns

2.3.1.1 Model-View-Controller

The Model-View-Controller (MVC) Architectural Pattern is designed to reduced coupling between system components, these are represented here as the user interface(s) (View) and the underlying data and its representation(s) (Model). The interaction(s) between the View and the Model “established using a subscribe and notify protocol” [8] (Controller). The Controller updates the View(s) based on the model(s) current state or vice-versa however; The View(s) cannot directly communicate with the Model, the Controller must govern such interactions.

This project will make effective use of Model-View-Controller in order to produce an environment which is much easier to maintain and has little coupling between the Model and the View(s). This allows the Model(s) and/or View(s) to be substituted or modified in order allowing different representations of the current algorithm, or in fact different algorithms altogether.

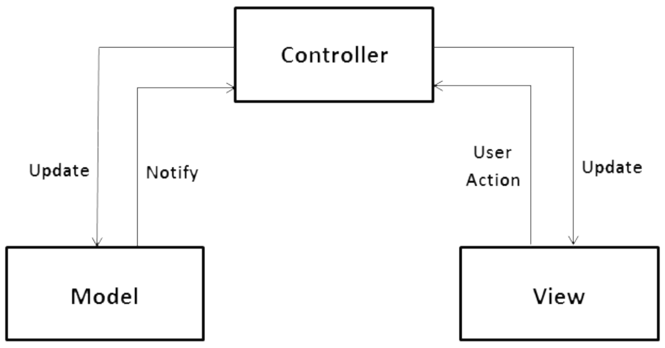


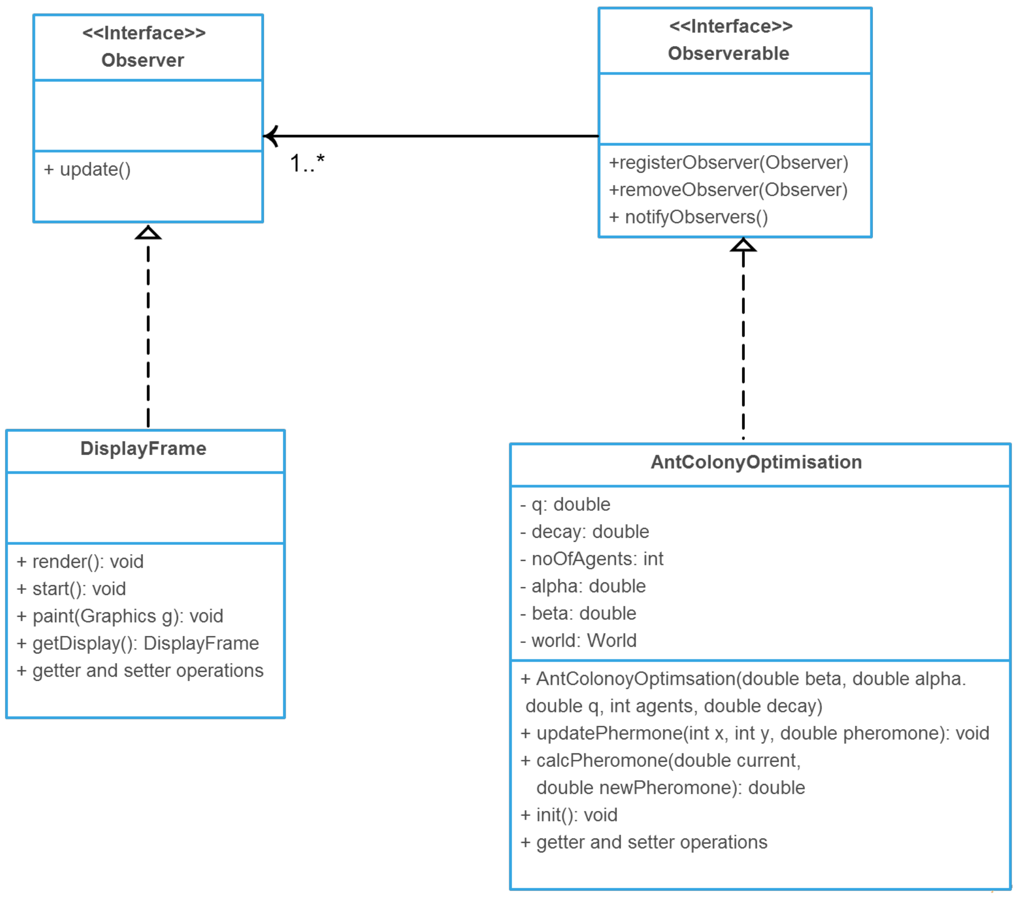
Figure B.1: Basic abstract overview of the Model-View-Controller pattern

- **Model** represents the underlying Ant Colony Optimisation algorithm. The Model also contains the required arithmetic functions and any additional operations required to execute the algorithm correctly.
- **View** represents the graphical user interface which will not only display the algorithms execution, but also enable the user(s) to modify the algorithms parameters.
- **Controller** represents the Observer required to enable interactions between the Model(s) and View(s).

2.3.2 Observer and Observable

The implementation of the Model-View-Controller design pattern is handled using the Observer and Observable interface provided as part of the default Java language specification. The general premise is that the Model will have an Observer which will have an update interface allowing it to receive signals from what it is observing (Model). The Model will implement the Observable interface which will enable it to send update signals to the Observer through the notifyObservers() method. This enables the Models dependencies to be updated as the Model itself changes ensuring the correct state of the Model is captured in the Views.

As figure B.2 demonstrates the View will Observe the Model and will wait for an update signal sent by the Model's notifyObserver() method. The Model can have multiple Observers using this pattern so there is the potential for future modification or enhancement without having to rework the existing system should the needs for extra views be needed.



<<Interface>>
Observer
+ update()
1..*
<<Interface>>
Observerable
+registerObserver(Observer)
+removeObserver(Observer)
+ notifyObservers()
DisplayFrame
+ render(): void
+ start(): void
+ paint(Graphics g): void
+ getDisplay(): DisplayFrame
+ getter and setter operations
AntColonyOptimisation
- q: double
- decay: double
- noOfAgents: int
- alpha: double
- beta: double
- world: World
+ AntColonoyOptimsation(double beta, double alpha. double q, int agents, double decay)
+ updatePhermone(int x, int y, double pheromone): void
+ calcPheromone(double current, double newPheromone): double
+ init(): void
+ getter and setter operations

Figure B.2: Proposed implementation of the Observer and Observable Design pattern

2.3.2.1 Singleton

In order to maintain simplicity throughout the application the Singleton design pattern will implemented for key Objects where one and only one instance of an Object must exist. The Singleton pattern prevents multiple instantiations of specific Object(s) as the Object itself is solely responsible for tracking the currently instantiated instance of itself [7].

The application will consist of a graphical user interface which in turn, will be composed of several different components. Such components must only be instantiated once in order to ensure correct interactions are performed. Without the presence of the Singleton pattern there exists the possibility of multiple instances of such components which could potentially cause unforeseen complications and undefined behaviours.

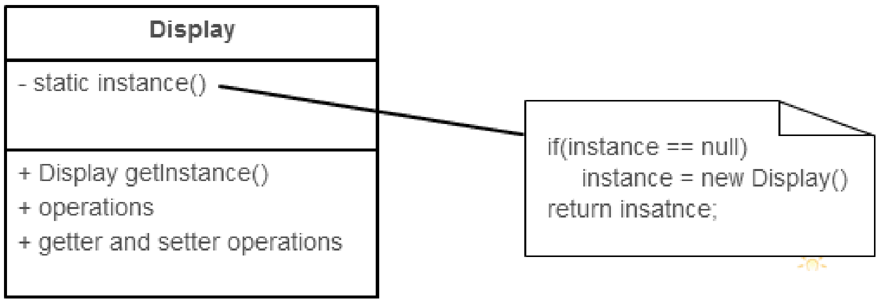


Figure B.3: Proposed high-level implementation of the Singleton pattern demonstrating how the sole instance of the graphical user interface will be tracked.

2.3.3 Structure

Adhering to the concepts of the patterns described in sections 2.3.1.1 and 2.3.2.1 the following Class Diagram shows proposed application structure. This is not concrete and could potentially change throughout development, the Class Diagram in question is represented below by figure B.4 and is represented using standard UML notation.

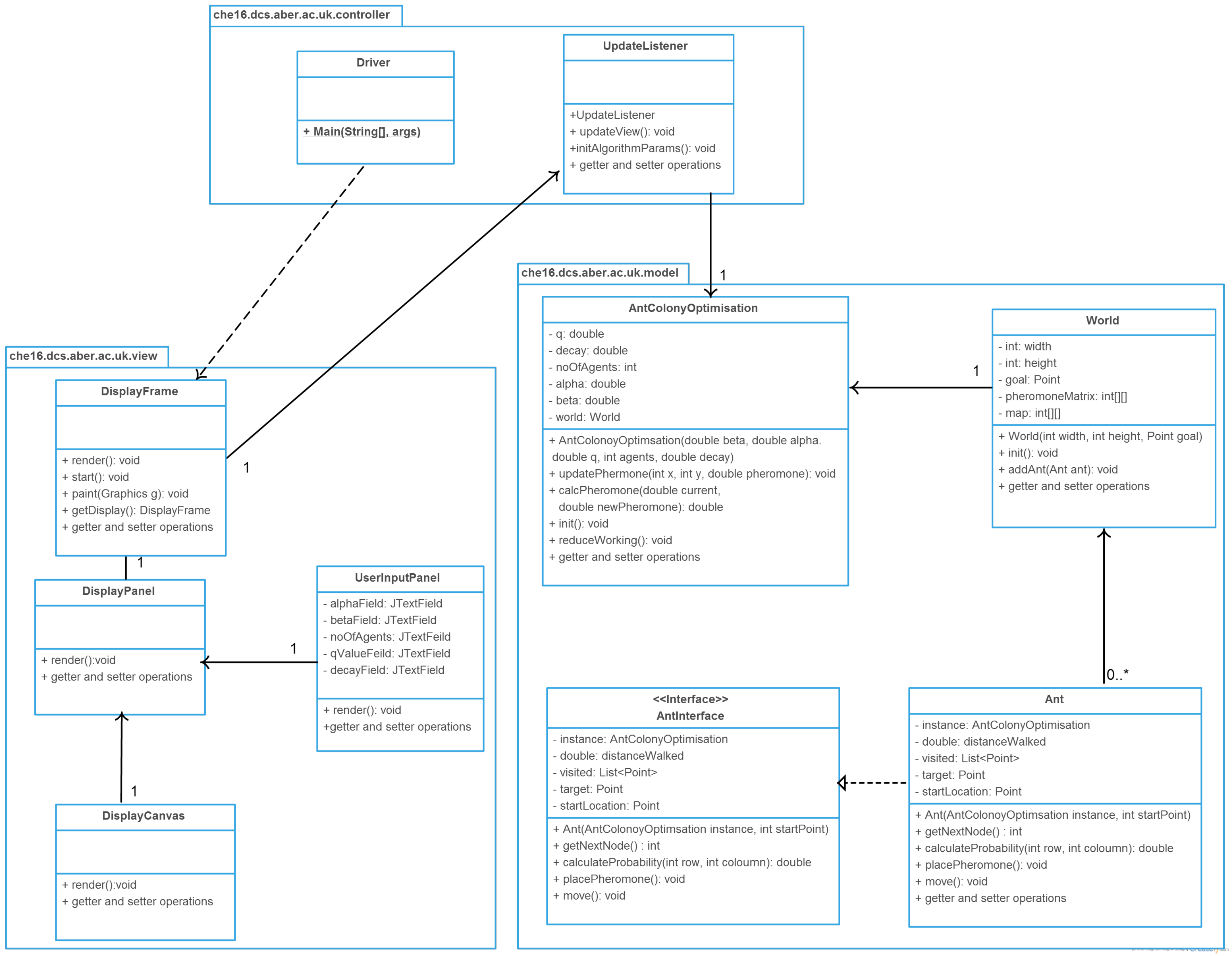

che16.dcs.aber.ac.uk.controller
Driver
+ Main(String[], args)
UpdateListener
+UpdateListener
+ updateView(): void
+initAlgorithmParams(): void
+ getter and setter operations
che16.dcs.aber.ac.uk.model
1
AntColonyOptimisation
- q: double
- decay: double
- noOfAgents: int
- alpha: double
- beta: double
- world: World
+ AntColonyOptimsation(double beta, double alpha.
double q, int agents, double decay)
+ updatePhermone(int x, int y, double pheromone): void
+ calcPheromone(double current,
double newPheromone): double
+ init(): void
+ reduceWorking(): void
+ getter and setter operations
1
World
- int: width
- int: height
- goal: Point
- pheromoneMatrix: int[][]
- map: int[][]
+ World(int width, int height, Point goal)
+ init(): void
+ addAnt(Ant ant): void
+ getter and setter operations
0..*
<<Interface>>
AntInterface
- instance: AntColonyOptimisation
- double: distanceWalked
- visited: List<Point>
- target: Point
- startLocation: Point
+ Ant(AntColonoyOptimsation instance, int startPoint)
+ getNextNode() : int
+ calculateProbability(int row, int coloumn): double
+ placePheromone(): void
+ move(): void
Ant
- instance: AntColonyOptimisation
- double: distanceWalked
- visited: List<Point>
- target: Point
- startLocation: Point
+ Ant(AntColonoyOptimsation instance, int startPoint)
+ getNextNode() : int
+ calculateProbability(int row, int coloumn): double
+ placePheromone(): void
+ move(): void
+ getter and setter operations
che16.dcs.aber.ac.uk.view
DisplayFrame
+ render(): void
+ start(): void
+ paint(Graphics g): void
+ getDisplay(): DisplayFrame
+ getter and setter operations
1
1
DisplayPanel
+ render():void
+ getter and setter operations
UserInputPanel
- alphaField: JTextField
- betaField: JTextField
- noOfAgents: JTextFeild
- qValueFeild: JTextFeild
- decayField: JTextFeild
+ render(): void
+getter and setter operations
1
1
DisplayCanvas
+ render():void
+ getter and setter operations

Figure B.4: Initial Class Diagram representing the main modules, packages and system interactions using UML notation.

2.3.3.1 View Package

The $che16.dcs.aber.ac.uk.view$ package contains the graphical user interface components. The main concept behind this is the use of nested components such as JPanel, JTextFields and the like. These will be contained inside top level JFrame. This allows modification of each component in isolation without impacting the other components behaviours and/or elements. This is the first module of the application which is initialised, and is done so directly from the $main$ method. This package has no direct knowledge of any package members mentioned in 2.3.3.2. Instead the $che16.dcs.aber.ac.uk.controller$ package members handle the communication(s) between the model(s) and view(s), adhering to the principles discussed in section 2.3.1.1.

2.3.3.2 Model Package

The che16.dcs.aber.ac.uk.model package contains the necessary elements and attributes required in order to accurately represent the algorithms state(s) and ensure correct algorithm execution. This package has no directl knowledge of any package members mentioned in 2.3.3.1. Instead the controller package members handle the communication(s) between these the model(s) and view(s), adhering to the principles discussed in section 2.3.1.1.

2.3.3.3 Controller Package

The initial concept for the controller is basic and will grow in complexity during the development lifecycle as more and more control based mechanisms will become prevalent. This initial concept is a simple Observer which notifies the view(s) should the model change in a significant way; the inverse of this is also true.

2.4 Class Descriptions

The Classes defined in Figure B.4 are described textually below.

2.4.1 Controller

2.4.1.1 Driver

The Driver Class is fairly simplistic and exists as an entry point to the application. This will Class handles any necessary instantiations and correctly assigns any key associations.

2.4.1.2 UpdateListener

The UpdateListener Class serves as a point of control for the update operations. The main updates will be handled internally by the relevant Classes (as described by Figure B.2) however, this Class will handle the updates caused when a user interacts with the application through the user interface. This includes, but is not limited to the modification of parameters or starting the algorithm's execution.

2.4.2 View

2.4.2.1 DisplayFrame

An instance of the DisplayFrame Class will serve as the highest level container for the applications user interface elements. This Class will be responsible for controlling access to, and the instantiation of the other graphical user interface elements.

2.4.2.2 DisplayPanel

The DisplayPanel instance will be contained inside the DisplayFrame. The purpose of the DisplayPanel is to contain the DisplayCanvas. The reason this Class is used to contain the DisplayCanvas rather than using the top level DisplayFrame is because it gives the author much more control over dimensions and modification of layouts or contents.

2.4.2.3 DisplayCanvas

The DisplayCanvas will be used to display the algorithms execution. This is the user interface element which will be painted and modified during the course of the algorithms execution.

2.4.2.4 UserInputPanel

The UserInputPanel is used to represent the collection of text boxes and buttons which will be used by the user to directly modify the algorithms parameters and state of execution. This will be contained in the top level DisplayFrame.

2.4.3 Model

2.4.3.1 AntColonyOptimisation

The AntColonyOptimisation Class is used to represent the high level behaviours of Ant Colony algorithms. This Class will contain the current instance of the World as well as housing all relevant data such as current parameter values and ways to interact with such parameters.

2.4.3.2 AntInterface

This interface will contain the necessary methods that every type of Ant (agent) has to have. This is by no means concrete, if there is only a need for one type of Ant then an interface is not strictly needed thus, it may be removed to reduce complexity.

2.4.3.3 Ant

An instance of the Ant Class will be used to represent an Ant (agent) in the world. As these are the agents which will be deployed on the world they will need to know about the World, whilst also knowing information about themselves including where they have visited so far and how far have they travelled. This Class will also contain the operations necessary in order for an Ant to be able to solve the current problem, this is main down the combination of knowing about the World and necessary details inside the move() method which will be used to select the Ants next location.

2.4.3.4 World

The World Class is a representation of the problem representation. This includes all information needed to model the graph as well as containing and manipulating the current pheromone matrix and pheromone concentrations on each edge. The World will be in constant communication with the Ant instances in order to communicate the current state of World to each agent.

2.5 Interface Design

2.5.1 Main User Interface

The interface design must accommodate all the necessary elements required to allow user defined values for each of the algorithm parameters. The interface must also be able to visually represent the current algorithms state including the locations of the agents and nodes without impacting on the usability of the application (the interface must not freeze or be negatively affected by the algorithms execution).

The interface will use a very neutral colour scheme which will maximise the usability and reduce the risk of complications which may arise from users being subject to difficulties understanding or identifying certain colours. Every option for the user will be textually defined and will not rely on any sound or visual prompts in order to user. As a result in addition to the above, users with visual impairments will still be able to use the application as intended.

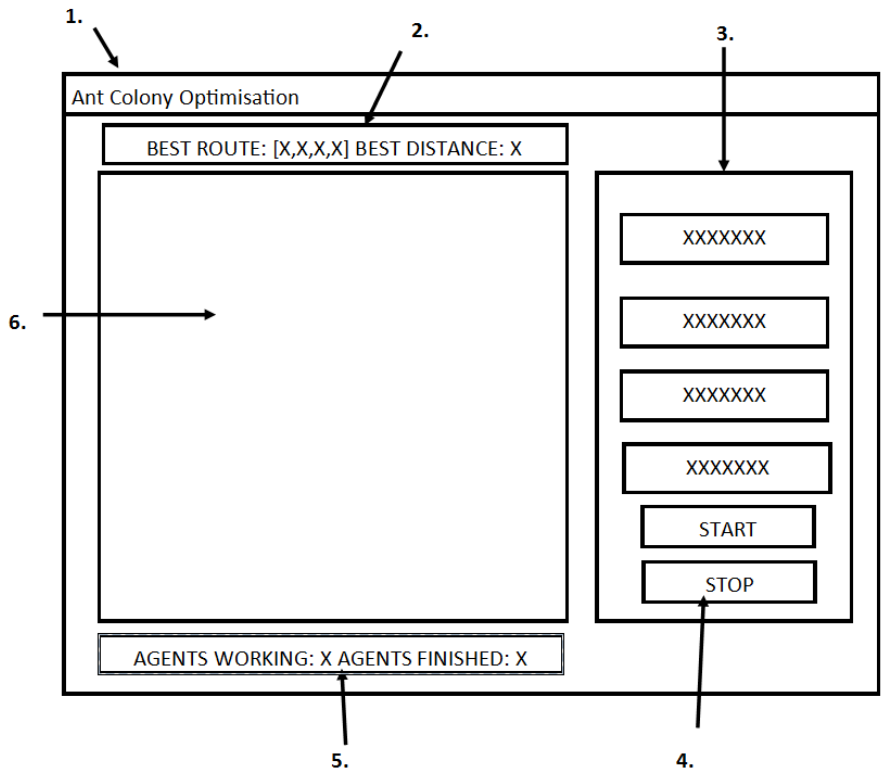
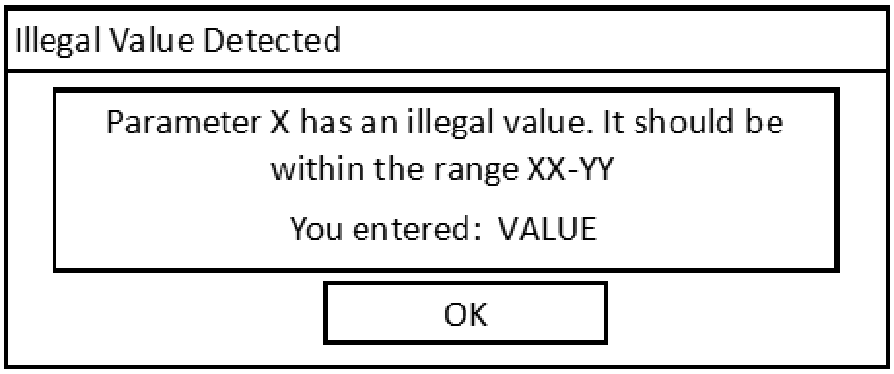

1.
2.
3.
4.
5.
6.
Ant Colony Optimisation
BEST ROUTE: [X,X,X,X] BEST DISTANCE: X
XXXXXXX
XXXXXXX
XXXXXXX
XXXXXXX
START
STOP
AGENTS WORKING: X AGENTS FINISHED: X

Figure B.5: Proposed design for the graphical user interface, showing the key elements in their planned locations.

1. in figure B.5 refers to the containing frame which will house the reset of the interface elements. This frame will not be resizable preventing complications such as the dynamic resizing of the interface having an impact on the observable range of the world. **2.** in figure B.5 refers to a text view containing information relevant to the algorithms current state of execution. This text view will contain the current best distance as well as the order of nodes visited to traverse the current best path. This will only be populated during the algorithms execution and only if the best path has been initialised. **3.** in figure B.5 refers to container which houses the interactive elements relating to the modification of the algorithms parameters. This will consist of several labels and text fields which can be modified informing the user of what they will be modifying as well as providing suitable error messages if the users enters an incorrect value for any of the parameters. **4.** in figure B.5 refers to the start and stop buttons. These are here to conform to the expectation that a user will expect some clear way to start and stop the application at their free choice, this is the simplest way to do this, and requires no hidden menus or hidden key bindings. **5.** in figure B.5 refers to a text view containing information relevant to the algorithms current state of execution. This text view will contain the number of agents currently working (which means the number of agents who haven't met their own stop conditions) as well as displaying the number of agents which are finished. **6.**in figure B.5 refers to the main canvas area which will display the algorithms current state of execution to the user. The contents of this will reflect the user defined values (elements in: **3.** in figure B.5) as well as representing each agents current location, their movements between nodes and also the modelling of pheromone deposit and decay will be present in this canvas.

2.5.2 Error Message Feedback

As the Algorithm parameters will be user defined using the interface proposed in section 2.5.1, figure B.5 there must be measures in place to catch and inform the user of any illegal values. Not only must the user be told they have inputted illegal values the illegal parameter value will be identified and a range of legal values will be displayed to the user.


Illegal Value Detected

Parameter X has an illegal value. It should be within the range XX-YY

You entered: VALUE

Figure B.6: Proposed design for the illegal parameter error message displayed to the user

As represented in figure B.6 the error message will contain all the information a user needs to identify the problem with their data input. The X value in figure B.6 represents which parameter has the illegal data input (i.e. Alpha, Beta). Range XX-YY represents the legal values for said parameter, and VALUE represents the value the user has specified for this parameter. The combination of these will provide the user with the knowledge of why this error message is being displayed and how they can resolve their issue.

This kind of error message will be composed using the JOptionPane and JDialog interfaces provided as default by the Java language specification. These views are very customisable and the styling will be handled by the languages underlying protocols which will reduce the codes complexity.

2.6 Algorithm

There are several adaptations of the Ant Colony Optimisation algorithm. Initially the project contain an implementation of Ant Colony Optimisation in its simplest form, without the presence of any enhancements such as using *Elitist Agents* or similar. Once a working implementation is in place the next step will be to adapt the algorithm in various ways to further aid the teaching potential of the project.

The general premise is that each Agent (Ant) embarks and a pseudo random walk through the state space. The Agents movements are influenced by pheromone deposits placed on edges between vertices. This pheromone is deposited by other Agents in accordance with the equations stated in section 2.6.2.2. The Agent's next move is influenced by the result of the equation stated in section 2.6.2.1. However; there is still the probability of the Agent moving to a less attractive point so the Agent does not always travel to the strongest pheromone concentration.

Overtime the pheromone deposit concentration on $edge_{xy}$ is directly proportional to the quality of the candidate solution, and ultimately the ants will converge to find the shorted route between two or more points.

There are several algorithm requirements;

- **Suitable problem representation** the World and Agents must be represented in a suitable and logical manner allowing the algorithm to execute as expected.
- **Pheromone manipulation metrics** there must exists adequate ways to access the pheromone matrix as well as manipulate (deposit/remove) the concentration of pheromone on a given edge in order to model decay and deposits.
- **Probabilistic movement functions** there must exist functions that calculate the probability of the Agent moving to a specific vertex. This is based on the pheromone concentration and the Agents location (see section 2.6.2.1).

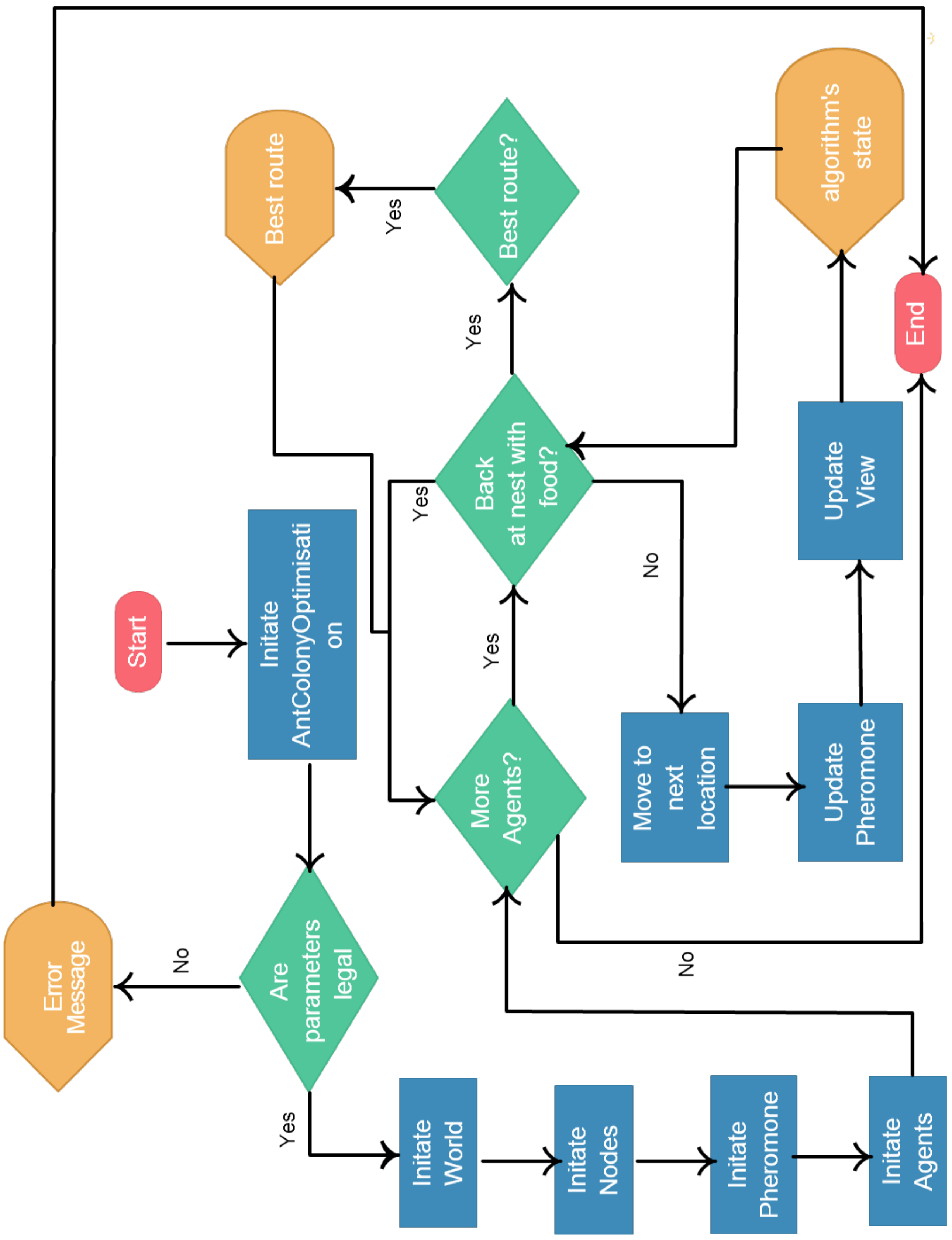


Figure B.7: High level flow digram representing the psudeo code for Algorithm 5, section 2.6.1

2.6.1 Pseudo-code

Algorithm 5 Pseudo-code for Ant Colony Optimisation

```
 1: Initiate AntColonyOptimisation with defined parameters
 2: if !$parameters\ are\ legal$ then
 3:     Dispaly error message to user
 4:     $return$
 5: Initiate World with algorithm parameters
 6: Initiate $Nodes$ and graph
 7: Initiate pheromone values
 8: Initiate $Agents$
 9: while !$all\ agents\ finished$ do
10:     for all Agents do
11:         while !back at $nest$ with $food$ do
12:             Calculate next move using probabilistic function
13:             Add moved point to Agent's memory
14:             Calculate and deposit pheromone on the path
15:             Update the View
16:         end while
17:     end for
18: if local best solution $<$ global best solution then
19:     $globalbest = local\ best\ solution$
20: end if
21: end while
22: output global best solution
```

Above is a somewhat simplified pseudo-code representation of the proposed Ant Colony Optimisation algorithm. The mathematical formulae required to achieve steps *7* and *10* are shown in section 2.6.2. The final algorithm may differ from the above depending on any additional feature present in the final release.

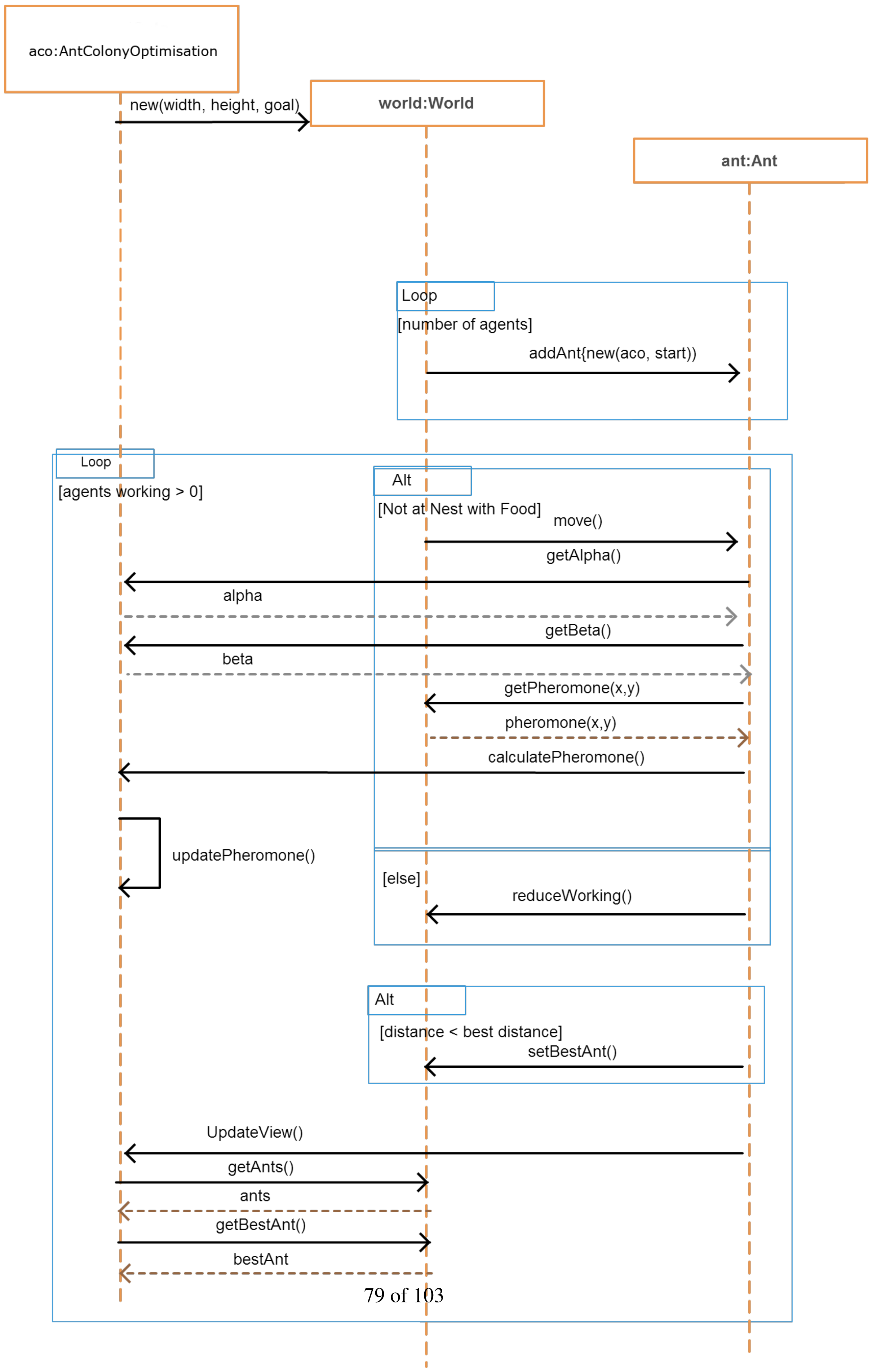


Figure B.8: Sequence diagram representing Algorithm 5, section 2.6.1

The sequence diagram shown in figure B.8 shows the high level interaction between the proposed main Model Classes during the algorithms execution. The Ants are in constant communication with the AntColonyOptimisation and World instances during their movement through the graph to ensure the correct pheromone values are being parsed during the movement process. The View is also updated through the updateView() method in the AntColonyOptimisation instance which requires data about the current Ants and the current best Ant. This data is stored in and returned from the World instance.

2.6.2 Metrics

2.6.2.1 Probabilistic function

$$p^k_{xy} = \frac{(\tau^\alpha_{xy})(\eta^\beta_{xy})}{\sum(\tau^\alpha_{xy})(\eta^\beta_{xy})} \tag{1}$$

Figure B.9: Algebraic model of the probabilistic function used to calculate next move for any Agent [25]

The probability that an Agent moves to vertex xy is described using the above. p is the probability for any Agent k to move through vertex xy. τ is the amount of pheromone deposited on vertex xy, which is raised to the exponent α. α is an heuristic value representing how greedy the algorithm is in its path finding [22]. The result for τ^α_{xy} is then multiplied by the edge xy's evaluation(η). Generally η will be represented using $\frac{1}{Euclidean\ distance_{xy}}$ [22] [20]. This will then be raised to the exponent β which like α is an heuristic parameter however β describes the Agents path finding speed.$\sum(\tau^\alpha_{xy})(\eta^\beta_{xy})$ is the sum of all possible solutions.

Algorithm 6 Pseudo-code for Probabilistic function - Each Agent - figure B.9

```
1: read the $pheromone$ level for vertex $xy$
2: raise the value from 1: to the exponent $\alpha$
3: Multiply the result of 2: by $(inverted\ distance_{xy})^\beta$
4: initiate a temporary double $cloumnTotal$
5: **for all** non-visited vertex **do**
6:     $columnToal$ += (read the $pheromone$ level for vertex $xy)^\alpha\times$ $(inverted\ distance_{xy})^\beta$
7: **end for**
8: divide the result of $3:$ by the result of $5:-7:$
```

2.6.2.2 Pheromone deposit

$$p^k_{xy} = (1-\rho)\tau^k_{xy} + \Delta\tau^k_{xy} \tag{2}$$

Figure B.10: Algebraic model of the pheromone deposit function used to calculate the correct values for the pheromone matrix [26]

τ represents the pheromone deposit for an edge xy by Agent k [22]. ρ is a value between $0 - 1$ which represents the decay rate $decay$. $1 - \rho$ is multiplied by the existing amount of pheromone at $edge_{xy}$ to correctly account for decaying trails. The new amount of pheromone is then added using the equation from figure B.11.

$$\Delta\tau_{xy}^{k} = \begin{cases} Q/L_k & \text{if Agent } k \text{ uses curve } xy \text{ in its tour} \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

Figure B.11: Algebraic model of the function used to calculate the how much new pheromone is deposited at xy [24]. Pheromone is only updated if the Agent k has visited point xy.

figure B.11 represents the new amount of pheromone to be added to the existing concentration at $edge_{xy}$. This can be read as change in τ ($\Delta\tau$). Q is simply another heuristic parameter which is divided by the distance $agentk$ travelled to get to $edge_{xy}$. If the result of this is ≤ 0 return 0. This ensures that new $pheromone$ is only added to the existing concentration at $edge_{xy}$ is used by $agentk$ in its tour.

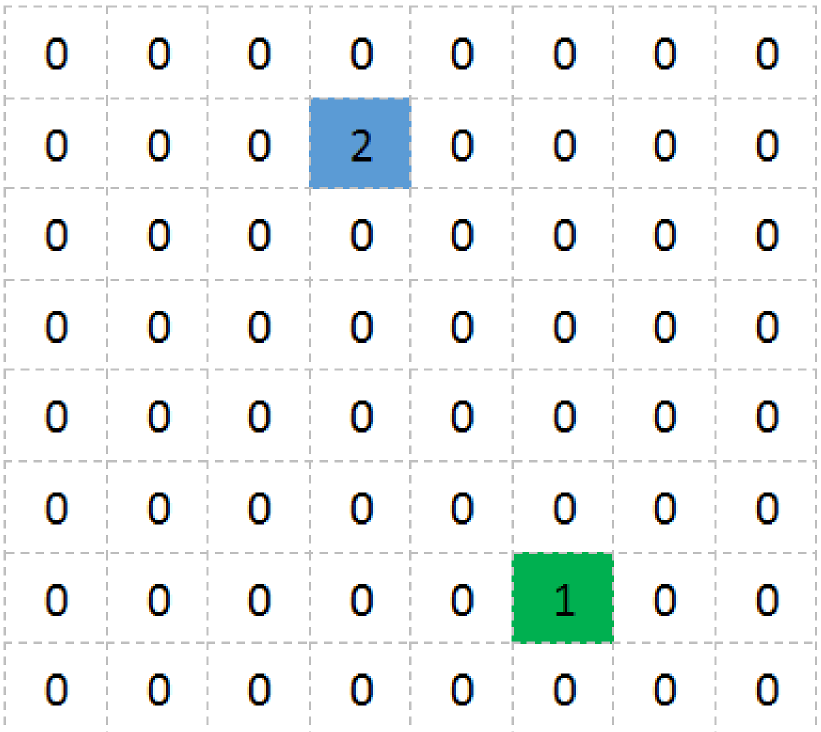
Algorithm 7 Pseudo-code for Pheromone function - figures B.10, B.11

```
1: if $pheromoneDeposit = Q\ value/totalDistanceWalked$ ¡ 0 then
2:     $pheromoneDeposit = 0$
3: $pheromone_{xy}$ = (1 - $algorithm\ decay\ rate$) $\times$ $currentPheromone_{xy}$ + $pheromoneDeposit$
4: if $pheromone_{xy} \geq 0$ then
5:     $pheromoneMatrix_{xy} = pheromone_{xy}$
6: else
7:     $pheromoneMatrix_{xy} = 0$
8: end if
```

2.7 Representation

2.7.1 World

The World Class is used to model the graph that the Ants will traverse during the algorithms execution. The graph will be represented as a two-dimensional Array, with each element in said array representing a Node in the graph itself. The two dimensional array will be composed of integer values with each element containing an integer value representing the terrain type at this index, this will also be used to set the nest and food locations.



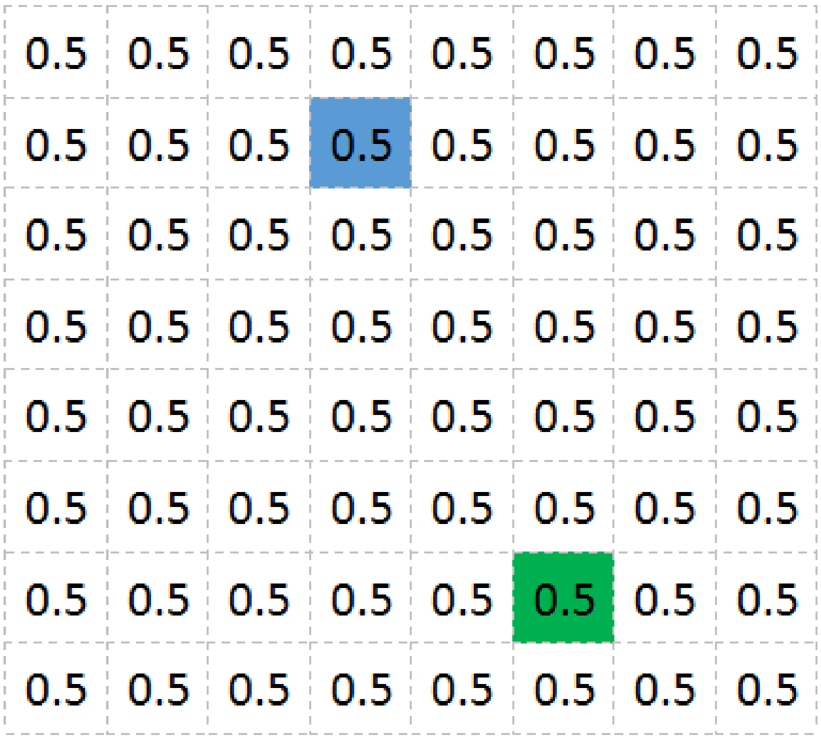
0	0	0	0	0	0	0	0
0	0	0	2	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0

Figure B.12: Example World representation using a two-dimensional Integer Array

Figure B.12 shows how the proposed graph representation would be implement. A ‘0’ value represents normal terrain where the Ants can move to, ‘1’ represents the nest location where the Ants will start and ‘2’ represents the food location which is the initial target for the Ants. This is a simplistic way to model an environment for the Ants whilst also allowing a multitude of terrain options without having to modify the way the graph is stored.

2.7.2 Pheromone

The pheromone must be modelled in a way which is both easily accessible and modifiable. The data structure used to store the pheromone must also relate to the graph mentioned in 2.7.1 to allow for logical mapping between the representation of the Ants environment and the pheromone associated with each Node in the graph. The pheromone will be stored as a two-dimensional Array of Doubles. Each element in the Array will represent the pheromone concentration for the corresponding Node in the graph for example, the double value at pheromone[x][y] will represent the pheromone concentration on edge [x][y] in the graph, thus there is a logical link between the two representations.



0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5

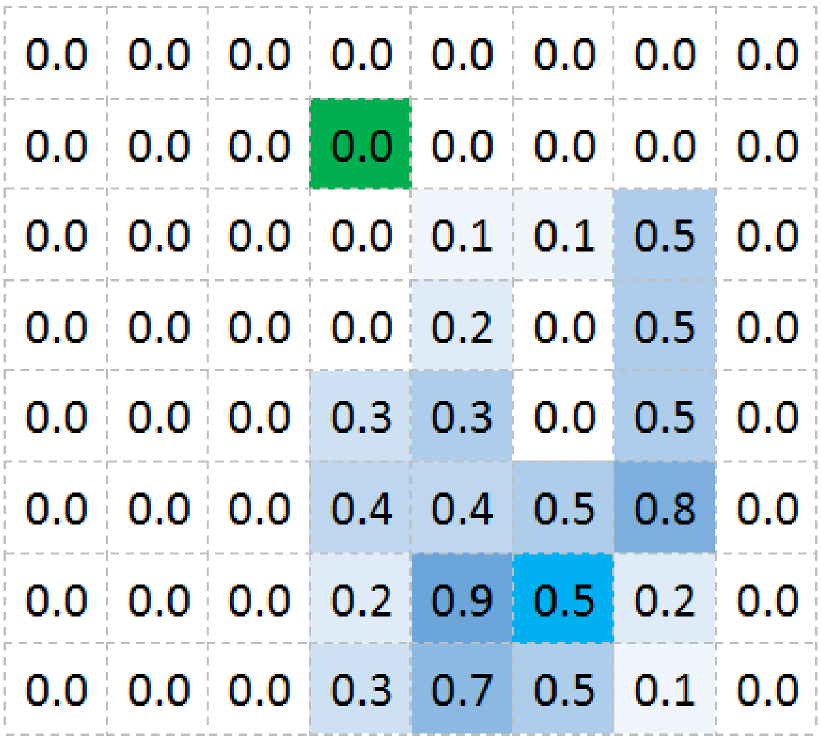
Figure B.13: Example Pheromone representation using a two-dimensional Double Array with default initial values

Figure B.13 demonstrates how a two-dimensional Double array can be effectively used to model the pheromone values for every Node in the graph. The values in this figure are default initial values which will be user defined when the application is executed. As the Ants moves through the graph the pheromone values at each index will correctly model the pheromone operations mentioned in section 2.6.2.2. As the values are stored in an array accessing or modifying the values is extremely simple and involves a simple getter or setter operation.

2.7.3 Visualisation

The data structures described in sections 2.7.1 and 2.7.2 must be visualised to the user in a manner that anybody can understand. Given that the graph will be stored as a two-dimension array it is perfectly logical to display this graph to the user in a grid format similar to that shown in figure B.12. The challenge with the visualisation process is visualising the pheromone deposit and decay operations as well as displaying the Ants moving between Nodes.

Given that the pheromone is stored in a two-dimensional array of doubles and the fact that accessing these values is extremely simple the value at each index can be used to directly influence how the user sees the pheromone. If each cell in the grid is coloured, and this colour's opacity is directly representational of the value at the corresponding index in the pheromone matrix then it becomes an accurate way of displaying the values of the pheromone to the user, which also allows the decay and deposit operations to be shown.



0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.1	0.1	0.5	0.0
0.0	0.0	0.0	0.0	0.2	0.0	0.5	0.0
0.0	0.0	0.0	0.3	0.3	0.0	0.5	0.0
0.0	0.0	0.0	0.4	0.4	0.5	0.8	0.0
0.0	0.0	0.0	0.2	0.9	0.5	0.2	0.0
0.0	0.0	0.0	0.3	0.7	0.5	0.1	0.0

Figure B.14: Example Pheromone visualisation

Figure B.14 shows how the pheromone value for a given [x,y] co-ordinate has a direct influence on the opacity of the cells colour. In said figure the pheromone value at each index is multiplied by 100 to convert it to a percentage, which then becomes the opacity value for the given Node. However, if the pheromone values for each Node become extremely small during execution there must be a measure in place to covert the small values back into percentages using a larger scaling factor than 100.

Appendix C

Implemented Code

```
pheromone = new Pheromone[cities.size()][cities.size()];
  for(int i = 0; i < pheromone.length; i++){
    for(int j = 0; j < pheromone[0].length; j++){
      pheromone[i][j] = new Pheromone(aco.getInitialPheromone());
    }
  }
```

Figure C.1: Snippet of code used to inialise the pheromone Matrix. *cities* is a list of $City$ obejcts. *aco* is the instance of the $AntColonyOptimisation$ Class

```
distanceMatrix = new double[cities.size()][cities.size()];
  for(int i = 0; i < distanceMatrix.length; i++){
    for(int j = 0; j < distanceMatrix[0].length; j++){
      distanceMatrix[i][j] =
          Globals.calculateEuclidianDistance(cities.get(i).getX(),
          cities.get(i).getY(), cities.get(j).getX(),
          cities.get(j).getY());
    }
  }
```

Figure C.2: Snippet of code used to inialise the distance Matrix. *cities* is a list of $City$ obejcts.

```
invertedMatrix = new
   double[distanceMatrix.length][distanceMatrix[0].length];
  for(int i = 0; i < distanceMatrix.length; i++){
    for(int j = 0; j < distanceMatrix[0].length; j++){
      invertedMatrix[i][j] = invertValue(distanceMatrix[i][j]);
    }
  }
```

Figure C.3: Snippet of code used to inialise the inverted distance Matrix. $cities$ is a list of $City$ obejcts. $invertValuedistanceMatrix[i][j]$ returns $\frac{1}{distanceMatrix[i][j]}$

```
cities = new ArrayList<City>();
  Random r = new Random();
  for(int i = 0; i < numberOfCities; i++){
    // the (+1) is to stop cities having the index '0' which would
       cause them to half render out of view
    int x = r.nextInt(aco.getBoundaryX()) + 1;
    int y = r.nextInt(aco.getBoundaryY()) + 1;
    //make sure 2 cities can't spawn on top of each other
    for(City city: cities){
      if(x == city.getX() && y == city.getY()){
        x = r.nextInt(aco.getBoundaryX()) + 1;
        y = r.nextInt(aco.getBoundaryY()) + 1;
      }
    }
    cities.add(new City(x,y,i));
  }
```

Figure C.4: Snippet of code used to inialise the collection of $City$ obejcts.

```
Random r = new Random();
  ants = new ArrayList<Ant>(numberOfAnts);
  for(int i = 0; i < numberOfAnts; i++){
    int index = r.nextInt(cities.size());
    for(City c: cities){
      if(index == c.getIndex()){
        c.adjustAntsHere(1);
        break;
      }
    }
    ants.add(new Ant(this, index));
  }
```

Figure C.5: Snippet of code used to inialise the collection of Ant obejcts. $this$ refers to the current $World$ Object.

```
for(int i = 0; i < pheromone.length; i++){
    for(int j = 0; j < pheromone[0].length; j++){
      updatePheromone(i, j, pheromone[i][j].getNewPhero());
      //reset the new phero values
      pheromone[i][j].resetNewPhero();
    }
  }

public void updatePheromone(int x, int y, double newPheromone) {
  double phero =
     calculatePheromones(pheromone[x][y].getPheromoneValue(),
     newPheromone);
  //if phero is not negative then update the current concentration
  //if phero is negative then just set it as 0, you can't have
     negative phero on an edge
  if (phero >= 0.0d) {
    pheromone[x][y].setPheromoneValue(phero);
  } else {
    pheromone[x][y].setPheromoneValue(0.0d);
  }
}

public double calculatePheromones(double current, double newPheromone)
   {
  //we dont need to store the result in a temporary variable, just
     return the equation in place
  return ((1 - aco.getDecayRate()) * current + newPheromone);
}
```

Figure C.6: Snippet of code used to update the pheromone values for every edge. *aco* is the current instance of the $AntColonyOptimisation$ Class.

```
public final int getNextProbableNode(int y) {
  //This is an adapted version of a similar method provided by Thomas
     Jungblut found here: https://code.google.com/p/antcolonyopt/
  //create a location to store the probability for all next locations
  //this can then be easily accessed to return the next index for the
     ant's move
  //only do this if there is in fact locations to move to
  if (unvisited > 0) {
    int danglingUnvisited = -1;
    final double[] weights = new double[visited.length];
    double columnSum = 0.0d;
    for (int i = 0; i < visited.length; i++) {
      columnSum += calculateIndividualProbability(y, i);
    }
    //once we have the value for sum (which the sum off all solutions
       evaluation)
    double sum = 0.0d;
    for (int x = 0; x < visited.length; x++) {
      if (!visited[x]) {
        weights[x] = calculateTotalProbability(x, y, columnSum);
        sum += weights[x];
        danglingUnvisited = x;
      }
    }
    //if sum is 0 then return, as this will be used in a division it
       cannot be zero
    if (sum == 0.0d)
      return danglingUnvisited;
    /* We need to give each index of the probability weighting based
       on the result of calculateToalProbability
     * this result is then divided by the total sum of all
        probabilities (usually 1)
     * The result of this division is then put in the correct index
        in the matrix in order to get the probability weighting
     *
     */
    double pSum = 0.0d;
    for (int i = 0; i < visited.length; i++) {
      pSum += weights[i] / sum;
      weights[i] = pSum;
    }
    //nextDouble returns a value between 0 and 1, so this can be used
       to effectively select a 'random' index based on the weighted
       probability
    //provided virtually as is by Thomas Jungblut:
       https://code.google.com/p/antcolonyopt/
    //this is what makes the ant walk 'randomly'
    final double r = random.nextDouble();
    for (int i = 0; i < visited.length; i++) {
      if (!visited[i]) {
        if (r <= weights[i]) {
          return i;
        }
      }
    }
  }
  return -1;
}
```

Figure C.7: Snippet of code used to select the Ants next location. This is a modifiedf version of the code provided by Thomas Jungblut [22]

```
boolean reset = false;
for(int i = 0; i < iterations; i++){
     aco.setCurrentIteration(i);
     if(aco.getRunning()){
       ArrayList<Ant> ants = (ArrayList<Ant>)aco.getWorld().getAnts();
       antsWorking = aco.getNoOfAgents();
       if(reset){
         //for the next iteration re-init the ants and go again
         for(City c: aco.getWorld().getCities()){
           c.resetAntCount();
         }
         aco.getWorld().initAnts();
         ants = (ArrayList<Ant>)aco.getWorld().getAnts();
         reset = false;
         }
       while(antsWorking > 0){
         for(Ant ant: ants){
           if(!aco.getRunning()){
             return null;
           }
           if(!ant.getFinished()){
             ant.move();
             aco.reduceWorking();
           }else{
             //if an ansst is finished, decrease the counter
             antsWorking--;
           }
         }
           aco.getWorld().decayPhero();
       }
       reset = true;
     }
  }
```

Figure C.8: Snippet of code used to automatically execute the algorithm untill compeltion. *aco* is the current instance of the $AntColonyOptimisation$ Class.

```
stepIndex = 0;
if(currentIter < iterations){
     running = true;
     Ant ant = world.getAnts().get(stepIndex);
     if(!ant.getFinished()){
        ant.step();
        this.notifyCanvas();
        if(ant.getUnvisted() == 0){
           ant.setFinished(true);
        }
        }else{
        reduceWorking();
        stepIndex++;
     }
        if(stepIndex >= world.getAnts().size()){
        //reset the ants and index
        world.initAnts();
        for(City c: world.getCities()){
           c.resetAntCount();
        }
        stepIndex = 0;
        agentsWorking = noOfAgents;
        currentIter++;
     }
  }
```

Figure C.9: Snippet of code used to execute the algorithm on a step by step basis

```
public static double linearInterpolateX(double x1,double x2, double
    mu) {
  return(x1*(1-mu)+x2*mu);
}

public static double linearInterpolateY(double y1,double y2, double
    mu) {
  return(y1*(1-mu)+y2*mu);
}
```

Figure C.10: Snippet of code determine the cordinate values for both x and y for a given μ value.

```
City start = null;
City destination = null;
  for(Ant ant: agents){
    if(!ant.getFinished()){
      for(City c: cities){
        if(ant.getCurrentIndex() == c.getIndex()){
          if(antImage != null){
            g2.drawImage(antImage, (c.getX() * 20) - 15, (c.getY() *
               20) - 9, null);
          }else{
            g2.setColor(Color.GREEN);
            g2.fillOval((c.getX() * 20) - 5, (c.getY() * 20) - 5,
               10, 10);
          }
        }
        if(ant.getMovementTracker()[0] !=
           ant.getMovementTracker()[1]){
          if(c.getIndex() == ant.getMovementTracker()[0]){
            start = c;
          }
          if(c.getIndex() == ant.getMovementTracker()[1]){
            destination = c;
          }
          if(start != null && destination != null){
            g2.setColor(Color.CYAN);
            for(int i = 1; i < 5; i++){
              //cast to a double to stop integer behaviours rounding
                 down to 0
              double result = ((double)i)/5;
              //draw an ellipse2D every 1/5 the way along the line
              float y = (float)
                 (Globals.linearInterpolateY(start.getY(),
                 destination.getY(), result));
              float x = (float)
                 (Globals.linearInterpolateY(start.getX(),
                 destination.getX(), result));
              Ellipse2D movement = new Ellipse2D.Float((x * 20.0f) -
                 5, (y * 20.0f) - 5, 10, 10);
              g2.fill(movement);
              g2.draw(movement);
          }
        }
      }
    }
  }
  //reset the values
  start = null;
  destination = null;
```

Figure C.11: Snippet of code used to visualise an Ants movement. $g2$ is an instance of $Graphics2D$.

```
bestRoute = model.getWorld().getBestRoute();
  if(bestRoute != null){
    for(int i = 0; i < bestRoute.size(); i++){
      for(City c: cities){
        if(bestRoute.get(i) == c.getIndex()){
          if(i + 1 < bestRoute.size()){
            for(City c2: cities){
              if(bestRoute.get(i + 1) == c2.getIndex()){
                g2.setColor(Color.RED);
                //make the best line slightly thicker than the rest
                    so it stands out more
                g2.setStroke(new BasicStroke(2));
                g2.drawLine(c.getX() * 20, c.getY()*20, c2.getX() *
                    20, c2.getY() * 20);
              }
            }
          }
        }
      }
    }
  }
```

Figure C.12: Snippet of code used to display the best route to the user.

```
for(EliteAntData data: eliteAnts){
  System.out.println(data.getEliteRoute());
  best = data.getEliteRoute();
  double e = (1/4) * numberOfCities;
  for(int i = 0; i < best.size(); i++){
    if(i + 1 < best.size()){
      pheromone[best.get(i)][best.get(i+1)].addToNewPhero
      (pheromone[best.get(i)][best.get(i+1)].getNewPhero() + e);
    }
  }
}
```

Figure C.13: Snippet of code used to depsoit pheromone along the elite routes. An $EliteAntData$ Object is used to represent an Elite route

```
Random r = new Random();
while(numberOfUphill > 0){
  int index = r.nextInt(cities.size());
  City temp = cities.get(index);
  index = r.nextInt(cities.size());
  if(!(temp.getUphilRoutes().contains(index)) && (index !=
     temp.getIndex())){
    temp.addToUphil(index);
    numberOfUphill--;
    distanceMatrix[temp.getIndex()][index] =
        (distanceMatrix[temp.getIndex()][index] * 2);
    invertedMatrix[temp.getIndex()][index] =
        1/distanceMatrix[temp.getIndex()][index];
  }
}
```

Figure C.14: Snippet of code used to initialise uphill paths.

Appendix D

Testing

4.1 Test Code Examples

```
@Test
public void testValidationRulesAllowLegalAlphaValues(){
  double[] values = new double[]{2.5, -2.1, 4.999999999,
     -4.999999999, 4, -2, -1.65, 5, -5, 3, -3.21};
  for(int i = 0; i < values.length; i++){
    assertTrue("validaite should return true alpha is legal",
       aco.validate(values[i], 2.5, 0.2, 0.2, 10, 15, 3, 3));
  }
}
```

Figure D.1: Code representing the automated testing process for checking mutliple alpha parameter values.

```
@Test
public void testValidationRulesAllowLegalBetaValues(){
  double[] values = new double[]{2.5, -2.1, 4.999999999,
     -4.999999999, 4, -2, -1.65, 5, -5, 3, -3.21};
  for(int i = 0; i < values.length; i++){
    assertTrue("validaite should return true Beta is legal",
       aco.validate(2, values[i], 0.2, 0.2, 10, 15, 3, 3));
  }
}
```

Figure D.2: Code representing the automated testing process for checking mutliple beta parameter values.

```
@Test
public void testValidationRulesAllowLegalAgentsValues(){
  int[] values = new int[]{25, 13, 19, 48, 29, 23, 11, 8, 2, 33, 40,
     39, 6};
  for(int i = 0; i < values.length; i++){
    assertTrue("validaite should return true number of agents is
       legal", aco.validate(2, 2.5, 0.2, 0.2, values[i], 15, 3, 3));
  }
}

@Test
public void testValidationRulesShouldNotAllowIllegalAgentsValues(){
  int[] values = new int[]{-1, -12, 55, 100, 87, 69, 52, 99, -72,
     -12, -5, 58, 60};
  for(int i = 0; i < values.length; i++){
    assertFalse("validaite should return true number of agents is
       illegal", aco.validate(2, 2.5, 0.2, 0.2, values[i], 15, 3, 3));
  }
}
```

Figure D.3: Code representing the automated testing process for checking mutliple agent parameter values.

```
@Test
public void testTotalProbabiliyIsCalculatedToEqualOne(){
   double value = 1.1275;
   double result = ((Math.pow(value, world.getAlpha()) *
       (Math.pow(value, world.getBeta())));
   assertEquals(1.82221,result, 0.0001);

   value = 3.1275;

   double result2 = ((Math.pow(value, world.getAlpha()) *
       (Math.pow(value, world.getBeta())));
   assertEquals(299.2172,result2, 0.0001);

   value = 2.1983;

   double result3 = ((Math.pow(value, world.getAlpha()) *
       (Math.pow(value, world.getBeta())));
   assertEquals(51.337509,result3, 0.0001);

   value = 0.0213;
   //test one with lower values 10^-9 is the result
   double result4 = ((Math.pow(value, world.getAlpha()) *
       (Math.pow(value, world.getBeta())));
   assertEquals(0.00000000438427,result4, 0.000000001);

   double totalSum = result + result2 + result3 + result4;

   assertEquals(352.376919, totalSum, 0.0001);

   double p1 = result/totalSum;
   double p2 = result2/totalSum;
   double p3 = result3/totalSum;
   double p4 = result4/totalSum;

   //check that total probability sums up to 1.0
   assertEquals(1.0, p1 + p2 + p3 + p4, 0.0001);
}
```

Figure D.4: Code representing the automated testing process for checking mutliple agent parameter values.

4.2 Black-box Testing

4.2.1 File Content Examples

The below figure D.5 contains an example of a valid problem configuration. Each line of content refers to a different specific value. A correct and valid file should contain values in this order and format;

- double, the alpha parameter value between must be in the range of $-5.0\ to\ 5.0$

- double, the double parameter value between must be in the range of $-5.0 \ to \ 5.0$
- double, the decay rate value must be in the range of $0 \ to \ 1$
- double, the initial edge pheromone value must be in the range of $0 \ to \ 1$
- int, the number of agents must be between $1 \ and \ 50$
- int, the number of cities must be between $3 \ and \ 25$
- int. int, the X and Y coordinate for a City. There should be 1 line for the corresponding number of cities defined in the previous line
- int, the number of uphill routes to be generated this must be between $0 \ and \ 15$
- int, the number of iterations this must be greater than 0
- EOF, signifies that this is the end of the configuration content

```
0.2
2.0
0.2
0.8
30
cities
15
36 6
30 1
19 1
25 22
8 7
2 20
26 19
19 3
24 23
7 9
12 6
18 27
2 24
35 23
21 27
7
5
EOF
```

Figure D.5: Contents of a legal problem representation stored in a file.

Below is an example of one of the test files used to ensure that only legal configurations are accepted. This file is rejected because the number of agents is specified as 30sffs which is not a valid integer value. It is easier to reject this and tell the user to correct it than it is to extract the integer value from a malformed integer.

```
0.2
2.0
0.2
0.8
30sffs
cities
15
36 6
30 1
19 1
25 22
8 7
2 20
26 19
19 3
24 23
7 9
12 6
18 27
2 24
35 23
21 27
7
5
EOF
```

Figure D.6: Contents of an illegal problem representation stored in a file. The number of agents in line 5 is not an integer value (30sffs)

Below is another example of an illegal configuration. The contents of this file contains no value representing the number of agents.

```
0.2
2.0
0.2
0.8
cities
15
36 6
30 1
19 1
25 22
8 7
2 20
26 19
19 3
24 23
7 9
12 6
18 27
2 24
35 23
21 27
7
5
EOF
```

Figure D.7: Contents of an illegal problem representation stored in a file. The number of agents has not been defined.

4.2.2 Tests

Annotated Bibliography

[1] 99designs.com, "Unbreakable laws of user interaction, [online] Mar 2015," Available: http://99designs.com/designer-blog/2014/01/15/7-unbreakable-laws-of-user-interface-design/, accessed: 7th Aril 2015.

A list of laws and explanations of such laws regarding important user interaction styles and principles

[2] Apache, "Maven, [online] Feb 2015," Avaliable: https://maven.apache.org/what-is-maven.html, accessed: 11th Aril 2015.

An overview of how the maven framework can be used and an analysis of the features provided.

[3] T. Blachard, ""Development of an Autonomous, Tethered and Submersible Data Buoy," Master's thesis, Aberystwyth University, Wales, Computer Science Department, Penglais Campus, Aberystwyth University, Aberystwyth, Ceredigion, SY23 3DB., May 2011, pp.15-16 [Online]. Available: http://users.aber.ac.uk/ttb7/files/ttb7_diss.pdf.

Tom Blanchard's dissertation, referenced pages contain relevant information and rationale about a suitable process which suited my development needs. Permission from Tom has been granted to adapt his rationale

[4] c2.com, "eXtreme Programming for one, [online] Apr 2015," Avaliable: http://c2.com/cgi/wiki?ExtremeProgrammingForOne, accessed: 9th Aril 2015.

An informal insight into eXtreme Programming and how it can be applied to solo development projects.

[5] cleveralgorithms.com, "Berlin52.tsp solution image, [online] Apr 2015," Avaliable: http://www.cleveralgorithms.com/images/tsp2.png, accessed: 7th Aril 2015.

An image mapping one solution to the Berlin52.tsp problem.

[6] M. Dorigo, "Optimization, learning and natural algorithms (in Italian)," Ph.D. dissertation, Dipartimento di Elettronica, Politecnico di Milano, Milan, Italy, 1992.

Marco Dorigo PhD thesis containing information about Ant Colony methods. Marco first came up with this concept however this thesis is extremely detailed and too complicated for this project and is used merely as a reference point should a reader want further information.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley Professional, Nov 1994, pp. 127-128, [Online]. Available: http://www.worldcat.org/isbn/020163361.

This book is a very good reference point for design patterns in general, providing concise descriptions, examples and use cases. The book examples are implemented using C++, as this will be a Java based project the main concepts are still relevant as the languages aren't too diss-similar.

[8] ——, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley Professional, Nov 1994, pp. 4-6, [Online]. Available: http://www.worldcat.org/isbn/020163361.

This book is a very good reference point for design patterns in general, providing concise descriptions, examples and use cases. The book examples are implemented using C++, as this will be a Java based project the main concepts are still relevant as the languages aren't too diss-similar.

[9] eclipse, "Eclipse IDE for Java Developers," Avaliable: https://eclipse.org/downloads/packages/eclipse-ide-java-developers/lunasr1a, accessed: 13th Aril 2015.

A free IDE for Java development. This is the IDE selected for this projects development and provided built in support for all the necessary tools.

[10] GitHub, "Atom," Avaliable: https://atom.io/, accessed: 13th Aril 2015.

A free, Open source text editor provided by GitHub.

[11] M. Dorigo and T. St ützle, *Ant Colony Optimization* , 1st ed. Bradford Books, Jun. 2004, pp.73[Online]. Available: http://www.worldcat.org/isbn/0262042193.

Publication by the creators of the Ant Colony family of methods. This publication contains the majority of information relevant to understanding the interactions and mappings of the real ants against the virtual agents and algorithm interactions. Page 73 contains information about the Elitist Ant System ants its formula.

[12] ——, *Ant Colony Optimization* , 1st ed. Bradford Books, Jun. 2004, pp.2-3[Online]. Available: http://www.worldcat.org/isbn/0262042193.

Publication by the creators of the Ant Colony family of methods. This publication contains the majority of information relevant to understanding the interactions and mappings of the real ants against the virtual agents and algorithm interactions. Pages 2-3 contain information about the Double Bridges experiment.

[13] ——, *Ant Colony Optimization* , 1st ed. Bradford Books, Jun. 2004, pp.2-3[Online]. Available: http://www.worldcat.org/isbn/0262042193.

Publication by the creators of the Ant Colony family of methods. This publication contains the majority of information relevant to understanding the interactions and mappings of the real ants against the virtual agents and algorithm interactions.

[14] Marco Dorigo, "Swarm intelligence, [online] Mar 2015," Available:http://www.scholarpedia.org/article/Swarm_intelligence, accessed: 7th Aprl 2015.

Online swarm intelligence definition which is somewhat more detailed than usual however, this definition is provided by the Ant Colony Optimisation creator thus it seems very fitting

[15] Mono, "Getting Started — Mono , [online] Feb 2015," Avaliable: http://www.mono-project.com/docs/getting-started/, accessed: 13 Feb 2015.

Mono provide an open source implementation of the Microsoft .NET framework enabling the development of C Sharp across multiple platforms and environments. The Framework is based on ECMA standards for C Sharp so there is some reliability when developing C Sharp in different environments. However, based on research the implementation of certain features, specifically the newer features in the latest releases of the .NET framework do not always behave as expected. This causes development problems which can easily be avoided for this project.

[16] neos, "Berlin52 source file, [online] Apr 2015," Available:http://neos.mcs.anl.gov/neos/solvers/co:concorde/Berlin52_TSP.txt, accessed: 7th Aril 2015.

Contents of the Berlin52.tsp problem representation.

[17] Paul Bourke, "Interpolation Methods, [online] Mar 2015," Avaliable: http://paulbourke.net/miscellaneous/interpolation/, accessed: 30th March 2015.

An overview and explanation of several methods of interpolation. The method of interest here is Linear Interpolation which is used in the representation of the Ants movement between cities.

[18] Robert Martin, "Single Responsibility Principle, [online] Apr 2015," Avaliable: http://blog.8thlight.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html, accessed: 15th Aril 2015.

An overview of the Single Responsibility Principle. This includes the underlying concepts and why it makes is logical to adhere to such concepts.

[19] V. E. Sjoerd, ""Dynamic ant colony optimization for the travelling salesman problem," Master's thesis, Leiden University, The Netherlands, Leiden Institute of Advanced Computer Science (LIACS), Leiden University, Niels Bohrweg 1, 2333 CA Leiden, The Netherlands, Jul 2012, pp.11 [Online]. Available: http://www.liacs.nl/assets/2012-08SjoerdvanEgmond.pdf.

A Masters thesis looking into a hybrid Ant and Bee Colony algorithm for dynamic TSP problems. The thesis has some good concepts and background into the algorithm, however a lot of the information is highly theoretical.

[20] ——, ""Dynamic ant colony optimization for the travelling salesman problem," Master's thesis, Leiden University, The Netherlands, Leiden Institute of Advanced Computer Science (LIACS), Leiden University, Niels Bohrweg 1, 2333 CA Leiden, The Netherlands, Jul 2012, pp.7 [Online]. Available: http://www.liacs.nl/assets/2012-08SjoerdvanEgmond.pdf.

A Masters thesis looking into a hybrid Ant and Bee Colony algorithm for dynamic TSP problems. The thesis has some good concepts and background into the algorithm, however a lot of the information is highly theoretical.

[21] technologyuk.net, "Waterfall model, [online] Feb 2015," Avaliable: http://www.technologyuk.net/computing/sad/waterfall_model.shtml, accessed: 9th Aril 2015.

An image depicting the structure of the waterfall life cycle.

[22] Thomas Jungblut, "Ant Colony Optimization for TSP Problems , [online] Feb 2015," Avaliable: http://codingwiththomas.blogspot.co.uk/2011/08/ant-colony-optimization-for-tsp.html, Aug 2011, accessed: 14 Feb 2015.

Simplistic explanations of each function required. Break down of equations is also included in order to make them more digestible.

[23] vnschool.net, "Double Bridge Experiment Image, online] Apr 2015," Avaliable: http://www.vnschool.net/images/082010/Thuattoan/ThuatT_2.jpg, accessed: 7th Aril 2015.

Simple graphic representing the Double Bridge Experiment.

[24] Wikipedia, "Ant Colony Optimisation New Pheromone Function , [online] Feb 2015," Available: http://upload.wikimedia.org/math/6/d/b/6db065218c956a4a7af6da99aaeca5d1.png, accessed: 14 Feb 2015.

Simple graphic representing the algebra representation of the pheromone function required to update the pheromone matrix.

[25] ——, "Ant Colony Optimisation New Probability Function , [online] Feb 2015," Available: http://upload.wikimedia.org/math/6/d/b/6db065218c956a4a7af6da99aaeca5d1.png, accessed: 14 Feb 2015.

Simple graphic representing the algebra representation of the pheromone function required to select the Agents next movements.

[26] ——, "Ant Colony Optimisation Pheromone Function , [online] Feb 2015," Available: http://upload.wikimedia.org/math/e/1/3/e1320f5f72b21e5766dfa7e29b536883.png, accessed: 14 Feb 2015.

Simple graphic representing the algebra representation of the pheromone function required to update the pheromone matrix.