technical-guide.md

Modelling an Ant Colony - Technical Guide

Project Title	Modelling An Ant Colony
Student 1 Name	Kevin Thomas Cleary
Student 1 ID	16373026
Student 2 Name	James Edward O'Neill
Student 2 ID	16410652
Project Supervisor	Alistair Sutherland

Date: 16/05/2020

Table of Contents

1. Motivation

Our work on this project was inspired by previous explorations into genetic algorithms and the use of biomimicry in technology.

In the past, Ant Colony Optimisation (ACO) algorithms have been utilised to solve defined shortest route problems, most notably the Travelling Salesman problem. This has been relayed and reflected in real world applications such as the design of traffic systems. We wanted to adapt this use case and explore the capabilities of an Ant Colony Optimisation in an environment where not every destination had to be visited, but rather find the shortest path between two locations. This would require a deep understanding of previous ACO algorithms to see where changes could be made to optimise a new algorithm specific to this environment.

We were motivated by a real life ant colony's ability to use pheromones to find their food sources, and adapt their routes quickly when an obstacle would block their original route. We explored how to make our algorithm 'recalibrate' and create a new shortest path if an obstacle blocked the initial path.

This project allowed us to gain a deeper understanding of the process behind developing a genetic optimisation algorithm - a field we had an interest in. We put into practice the use of a defined test case where we knew what the optimum length was in order to test the quality of our algorithm. We researched previous probabilistic equations that such algorithms use to learn and choose paths to their benefit, what constants should be used in such equations and the values of these constants.

2. Research

2.1 Swarm Intelligence

Ant Colony Optimisation Algorithms examplify **Swarm Intelligence** (SI).

> Swarm Intelligence: The discipline that deals with natural and artificial systems composed of many individuals that coordinate using decentralized control and self organizaion.

Swarm intelligence focuses on the collective behaviours that result from the local interactions of all the individuals within a system with each other and their environment. In this case we focus on the behaviour of individual ants within a colony as they search for food.

A typical SI system has the following properties:

- Composed of many individuals
- Individuals are relatively homogeneous (they are either all identical or belong to a few typologies)
- Interactions among the individuals are based on simple behavioural rules that exploit only local information that the individuals exchange directly or via the environment (**stigmergy**)

- Overall behaviour of the system results from the interactions of individuals with each other and with their environment - group behaviour self organizes.

Swarm intelligence can be split up into **Natural** and **Artificial** research:

- Natural SI research studies biological systems; as well as ants, the behaviour of schools of fish, flocks of birds, termite colonies and herds of land animals are all examples of Natural Swarm Intelligence.
- Artificial SI focuses on human artifacts eg multi-robot systems.

Swarm intelligence is also divided into two streams; **Scientific** and **Engineering**:

- Scientific stream involves modelling swarm intelligence systems to single out and understand the mechanisms that allow a system as a whole to behave in a coordinated way as a result of local interactions.
- Goal of Engineering stream is to exploit the understanding developed by the scientific stream in order to design systems able to solve problems of practical relevance.

This project is in a **Natural** field, and is based around **both streams**; we are studying the effect of the local individual-individual and individual-environment interactions of our simulated ants on the behaviour on our colony system as a whole, while also applying our resulting algorithm to a real-world route recalibration application.

2.2 Ant Colony Optimisation Algorithm Overview

> Ant Colony Optimisation is a **population-based metaheuristic** that can be used to find approximate solutions to difficult optimization problems.

In ACO, a set of software agents called **artificial ants** search for good solutions to a given optimization problem. To apply ACO, the optimization problem is typically transformed into the problem of finding the best path on a weighted graph. The artificial ants (referred to henceforth as 'ants') incrementally build solutions by moving on the graph. The solution construction process is **stochastic** and is biased by a **pheromone model**, that is, a set of parameters associated with graph components (either nodes or edges) whose values are modified at runtime by the ants.

The **Construction Graph**, or Construction Matrix, is the virtual space in which the ants travel. The ants make probabilistic decisions based on pheromone concentrations at each point in the construction matrix.

An ACO is usually used to tackle a **Combinatorial Optimisation Problem** (COP). Marco Dorigo (IRIDIA 1997) defines the COP as a triplet S, Ω, f where:

- S is a **search space** defined over a finite set of discrete decision variables
    - a set of discrete variable X_i, i = 1, ..., n with values $v^j_i \in D_i = \{v^1_i, ..., v_i^{|Di|}\}$, is given
    - Elements of S are full assignments; in which each variable X_i has a value v^j_i assigned from its domain D_i.
- Ω is a **set of constraints** among the variables
    - Set S_Ω is given by the elements of S that satisfy all constraints in the set Ω
- $f : S \rightarrow R^+_0$ is an objective function to be minimised.
    - as maximising over f is the same as minimising by -f, any COP can be considered a minimization problem

A solution $s \in S_\Omega$ is called the global optimum if and only if:

- $f(s_*) \leq f(s)\ \forall s \in S_\Omega$

Set of all globally optimum solutions denoted by $S_\Omega \subseteq S_\Omega$. Solving a COP requires at least one $s \in S_\Omega$.

The Travelling Salesman Problem (TSP) is a well known example of a COP.

2.3 Ant Colony Optimisation Algorithm Metaheuristic

> In ACO, artificial ants build a solution to a Combinatorial Optimisation Problem by traversing a fully connected construction graph.

- Each instantiated decision variable $X_i = v^j_i$ is called a **solution component**, denoted by c_{ij}. Set of all possible solution components is denoted by C.
- The **construction graph** $G_C(V,E)$ is defined by associating the components C either with the set of vertices V or the set of edges E.
- A **pheromone trail value** τ_{ij} is associated with each solution component. Allows the probability distribution of different components of the solution to be modelled. Pheromone values are used and updated by the ACO algorithm during the search.

- Ants typically move from vertex to vertex across edges of the construction graph exploiting info provided by the numerous **pheromone trails**, as well as depositing their own. $\Delta\tau$, or the **amount of pheromones** deposited depend on the quality of the solution found.

The metaheuristic for an ACO algorithm is as follows:

```
Set parameters, initialise pheromone trails

SCHEDULE_ACTIVITIES
    ConstructAntSolutions
    DaemonActions {optional}
    UpdatePheromones
END_SCHEDULE_ACIVITIES
```

The metaheuristic consists of an **initialisation** phase and three **activity** phases executed repeatedly as a schedule of activities. These scheduled activities aren't specified in any given order.

2.3.1 ConstructAntSolutions

- A group of m ants construct solutions from elements of a finite set of available components C = {c_{ij}}, i = 1, ..., n, j = 2, ..., |D_i|.
    - A solution construction starts with an empty solution $s^p = \varnothing$.
    - At each construction step, current partial solution s^p is extended by adding a feasible solution component from the set of feasible neighbours $N(s^p) \subseteq C$.
- The choice of a solution component from $N(s^p)$ is done **probabilistically** at each construction step. Below is Dorigo's method where,
    - τ_{ij} is the pheromone value associated with c_{ij}
    - η_{ij} is the heuristic value associated with c_{ij}
    - α and β are positive real parameter whose values determine the relative importance of pheromone vs heuristic info

$$p(c_{ij}|s^p) = \frac{\tau_{ij}^{\alpha} \cdot \eta_{ij}^{\beta}}{\sum_{c_{il} \in N(s^p)} \tau_{il}^{\alpha} \cdot \eta_{il}^{\beta}}, \forall c_{ij} \in N(s^p)$$

2.3.2 DaemonActions

- Refers to any problem specific/centralised actions that cannot be performed by a single ant.
- The most used daemon action consists in the application of **local search** to the constructed solutions: the locally optimized solutions are then used to decide which pheromone values to update.

2.3.3 PheromoneUpdate

- The aim is to increase pheromone values associated with good solutions and to decrease pheromone values associated with bad solutions
- A common approach is to decrease all pheromone levels using pheromone **evaporation** and then increasing pheromone levels associated with good solutions by using **S_{upd}**
- $$\tau_{ij} \leftarrow (1-\rho) \cdot \tau_{ij} + \rho \cdot \sum_{s \in S_{\text{upd}} | c_{ij} \in s} F(s)$$
    - Where S_{upd} is a set of solutions used for the update, $\rho \in (0,1]$ is a parameter for the **evaporation rate** and $F: S \to R^+_0$ is a function such that:

      ```
      f(s) < f(s′) ⇒ F(s) ≥ F(s′), ∀s ≠ s′ ∈ S
      ```

    - F(.) is commonly referred to as the **fitness function**
- Pheromone evaporation implements a useful form of forgetting, favouring new areas in the search space
- This is where different systems tend to branch off from each other eg AntColonySystem (Dorigo & Gambardella) or MaxMinAntSystem (Stützle & Hoos)

- There exist different specifications of S_{upd}, usually a subset of $S_{iter} \cup \{ S_{bs} \}$
  - S_{iter} is the set of solutions constructed from the current iteration of the algorithm
  - S_{bs} is the best solution found since the first iteration AKA best so far
- Example 1: the **Ant System** Update System
  - $$S_{upd} \leftarrow S_{iter}$$
- Example 2: the **Iteration-Best** (IB) Update System
  - $$S_{upd} \leftarrow \arg\max_{s \in S_{iter}} F(s)$$
- The IB Update rule contains a much stronger bias towards good solutions than the AS update rule does. This increases the speed of finding a solution but also increases the probability of premature convergence.
- An even stronger bias can be found in the **Best-So-Far** (BS) update rule, where S_{upd} is set to $\{ S_{bs} \}$
- In practice it is best to use variations of the IB or BS rules which avoid premature convergence to acheive better results than an AS update rule.

2.4 History of popular ACO Algorithms

2.4.1 Ant System

Dorigo et. al, 1991-1996

> The Ant System (AS) has virtual ants construct solutions on a construction graph and updates pheromones between components on each iteration.

The first Ant Colony Optimisation Algorithm to be proposed in literature. Pheromone values are updated for all of the ants that have completed the tour. Solution components c_{ij} are the edges of the graph and the pheromone update for τ_{ij} (the pheromone joining nodes i and j) is as follows

- $$\tau_{ij} \leftarrow (1-\rho) \cdot \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k}$$
  - $\rho \in (0,1]$ is the evaporation rate of pheromone trails
  - *m is the number of ants
  - $\Delta\tau^{k}_{ij}$ is the quantity of pheromones laid on edge (i, j) by ant k
    - $$\Delta\tau_{ij}^{k} = \begin{cases} \frac{1}{L_k} & \text{if} \setminus \text{ant } k \text{ used} \setminus \text{edge } (i,j) \text{ in} \setminus \text{its} \setminus \text{tour}, \\ 0 & \text{otherwise}, \end{cases}$$
    - where L_k is the tour length of the k-th ant

When constructing their solutions, the ants traverse the construction graph making a probabilistic decision at each vertex. The **transition probability** of the k-th ant moving between nodes i and j is given by:

- $$p(c_{ij}|s_k^p) = \begin{cases} \frac{\tau_{ij}^{\alpha} \cdot \eta_{ij}^{\beta}}{\sum_{c_{il} \in N(s_k^p)} \tau_{il}^{\alpha} \cdot \eta_{il}^{\beta}} & \text{if } j \in N(s_k^p), \\ 0 & \text{otherwise}, \end{cases}$$
  - $N(s^p_k)$ is the set of components that do not yet belong to the partial solution s^p_k of ant k
  - α and β are parameters that control the relative importance of the pheromone versus the heuristic information $\eta_{ij} = 1/d_{ij}$, where d_{ij} is the length of the component c_{ij} (ie of edge (i,j))

2.4.2 Ant Colony System

Dorigo & Gambardella, 1997

> The Ant Colony System (ACS) introduces new random factor q and a local pheromone update rule to diversify solution construction.

First major improvement over the original Ant System. First innovations made in a system known as Ant-Q, by the same authors.

New additions include the form of the **decision rule** used by ants during the construction process. Ants in ACS use the **psuedorandom porportional** rule: the probability that an ant moves from node i to j depends on a random variable q uniformly distributed over [0,1], and a parameter q_0. If $q <= q_0$ then, among the feasible components, the component to maximise the product of $\tau_{il}\eta^{\beta}_{il}$ is chosen. Otherwise the same equation as AS is used.

This is a **greedy rule** which favours exploitation of the pheromone information, and must be counterbalanced with the introduction of a diversifying component, the **Local Pheromone Update** (LPU). Local pheromone update is performed by all ants after each construction step. Each ant only applies it to the last edge traversed.

- $\tau_{ij} = (1 - \varphi) \cdot \tau_{ij} + \varphi \cdot \tau_0$
  - Where $\phi \in (0,1]$ is the pheromone decay coefficient and τ_0 is the initial pheromone value.

The main goal of LPU is to diversify the search performed by subsequent ants during one iteration. Decreasing the pheromone concentrations on traversed edges in an iteration encourages other following ants to choose different edges and form different solutions. Local Pheromone Update also means the minimum values of the pheromone are limited.

An **offline pheromone update** is perfomed at the end of the construction process, as in AS. This is performed only by the best ant - only the edges visited by the best ant are updated according to the following equation

- $\tau_{ij} \leftarrow (1 - \rho) \cdot \tau_{ij} + \rho \cdot \Delta\tau_{ij}^{best}$
  - if the best ant used edge (i,j) in its tour, then $\Delta\tau^{best}_{ij} = 1/L_{best}$
  - L_{best} can be either the length of the iteration best or that of the best-so-far
  - otherwise, $\Delta\tau^{best}_{ij} = 0$

2.4.3 MAXMIN Ant System

Stützle & Hoos, 2000

> In MAXMIN ant system (MMAS), only the best ant adds pheromone trails, and the minimum and maximum values of the pheromone are explicitly limited (AS and ACS are implicitly limited)

The pheromone update equation takes the form

- $\tau_{ij} \leftarrow (1 - \rho) \cdot \tau_{ij} + \Delta\tau_{ij}^{best}$
- if the best ant used edge (i,j) in its tour, then $\Delta\tau^{best}_{ij} = 1/L_{best}$
  - L_{best} can be either the length of the iteration best or that of the best-so-far
- Pheromone values are constrained between τ_{min} and τ_{max} by verifying that each pheromone value after being updated by the ant is within these constraints
- τ_{ij} is set to τ_{max} if $\tau_{ij} > \tau_{max}$ and is set to τ_{min} if $\tau_{ij} < \tau_{min}$
- As in AS but contrary to ACS, the pheromone update rule applies to all edges at the end of an iteration rather than those traversed by the best ants.
- τ_{min} is usually experimentally chosen (Stutzle & Hoos have developed a theory on how to define this value analytically)
- τ_{max} can be calculated analytically if the optimum ant tour length is known
  - for TSP, optimum tour length would be $1 / (\rho \cdot L)$, where L is the length of the optimal tour.
  - L can be substitued with L_{bs} if L not known

The algorithm sets the initial values of its trails to L_{max} and stops when no improvement can be seen for a given number of iterations.

2.5 Adaptions to Ant Colony Optimisation Algorithm

The problem we are looking to solve with our system differs from the COPs tackled by the systems described above in various ways. As such, our design and implementation need to differ slightly from these systems based on the requirements of our problem.

The problem we wish to solve is as follows:

> Develop a system modelled on the behaviour of an ant colony to find the shortest route from the Home Node (an 'anthill') to a specified Goal Node (a 'food source') in a matrix, using pheromone value as a determining factor of an ants' trail.

Using the Travelling Salesman Problem (TSP) as a reference for a typical Combinatorial Optimisation Problem, we can identify the following differences:

- In a TSP, the objective is to find the shortest path around an **entire matrix** of differently weighted nodes, attempting to find the shortest path visiting every possible node and returning home.
    - Our own problem does not require that every node in the matrix is visited, but rather that every ant begins at the same node and should end at the same, sperate node.
- Following on from this, the solution set of an Ant System solving a TSP comprises of a **list of edges** between pairs of nodes within the construction matrix, and the shortest path is the list with the lowest sum of values.
    - For the purpose of our own problem and also our visual simulation, our solution set will comprise of a **list of nodes**, each equidistant from each of its neighbouring nodes. The shortest path will be the list with the smallest number of elements.
- When selecting a new node in all previous ACO systems, ants use a distance heuristic which grows in prominence the closer a solution is to construction. This is based on the number of nodes in the matrix left to be visited.
    - As our problem does not specify this number of nodes, we have to redefine this distance heuristic. We have decided to base our distance heuristic on the distance from the current ants position from the home node. Our probabilitic function favours nodes further away from the home node.

3. Design

Our main tasks from the beginning were to (i) create an algorithm that modelled an ant colony's path-finding abilities, and (ii) display how the algorithm works visually. These two tasks defined how the project would be designed, ie. in two parts; the Ant Colony Optimisation, and the Visual Simulation.

3.1 Ant Colony Optimisation

This part of the project was responsible for defining:

- how the ants move around the given environment,
- how they leave scent trails on the environment,
- how they avoid obstacles.

Once all of these are defined, the ants would be able to find the goal from their starting position.

3.1.1 Moving Around the Given Environment

In our project, unlike other path-finding algorithms, the ants have no prior knowledge of the environment they inhabit. The ants can only see their immediate surroundings, and the pheromone trails present in their immediate surroundings. Similar to simulated annealing, the ants have a certain probability that they will move in a random direction without considering the state of their immediate surroundings. This allows ants to find newer, possibly better paths.

If the probability of moving in a random direction is not satisfied, then the ant uses their immediate surroundings to determine which direction they will move in. The direction of the next move is influenced by two factors:

1. The ant's distance from its home, and
2. The amount of pheromones that are present.

If the ant is close to its home, then the direction that leads away from its home is more likely to be chosen. This prevents the ants from doing too much backtracking, and encourages further exploration of the environment.

If a direction contains a lot of pheromones, then that direction is more likely to be chosen by the ant. This allows ants to learn from other, successful ants.

In the beginning, there are no pheromones present in the environment, so the distance from the goal is the only influence on the ants' movement. It is only when there are successful ants that pheromone trails become a factor for choosing a direction. This causes the ants to move in a mostly random way for the first iteration of the algorithm.

3.1.2 Leaving Scent Trails (Pheromones)

Scent trails or pheromone trails are only present when there has been a successful ant; ie. an ant has reached the goal. Once an ant reaches the goal, the ant colony optimisation algorithm propagates pheromones along the trail of the successful ant. If an ant is unsuccessful in finding the goal by the end of the iteration, then no pheromones get distributed along its trail.

Each ant starts an iteration with the same number of pheromones, and once an ant reaches the goal, these pheromones get distributed evenly along the trail. This means that the longer the trail, the weaker the pheromones. This ensures that shorter trails receive higher amounts of pheromones, and as a result, are more preferred by subsequent ants.

3.1.3 Avoiding Obstacles

The obstacles represent an impasse in the environment. The ants are unable to pass through the obstacles, and this adds another layer of complexity to the simulation. This is where the random movements and pheromones really come into play. If no obstacles existed in the simulation, then ants would be able to find the shortest path to the goal with no issue, as it would just be a beeline from their home to the goal. Instead, they must use random movement to find new paths, or use pheromone trails to guide them in the correct direction.

3.2 Visual Display of the Algorithm (GUI)

The second part of our project was to visually display the algorithm to the user. The visual display of the algorithm is broken down into the following features:

1. Display the environment,
2. Display the ants' movements throughout the simulation,
3. Allow the user to start, stop, pause, and skip an iteration, and
4. Allow the user to specify various variables in the algorithm for the next time the simulation is run.
5. Allow the user to add obstacles to the environment.

These features are useful for understanding how the algorithm works, to test the algorithm, or to use the algorithm to find paths through a given environment.

4. Implementation

All of the source code for this project is written in Java 8. We thought that an object oriented design would be the best way to implement both the algorithm and the visual display, having classes for the environment, each ant, etc. Each component of the algorithm needs to hold state throughout the running of the algorithm, so an object oriented implementation seemed the best choice.

The implementation of our project follows and extends the design, with the two main components being included alongside the means of running the simulation and various scenarios, the means of testing the algorithm, gathering metrics on these tests, and displaying these metrics so that the optimum state of the variables in the `AntColonyOptimisation` algorithm can be found.

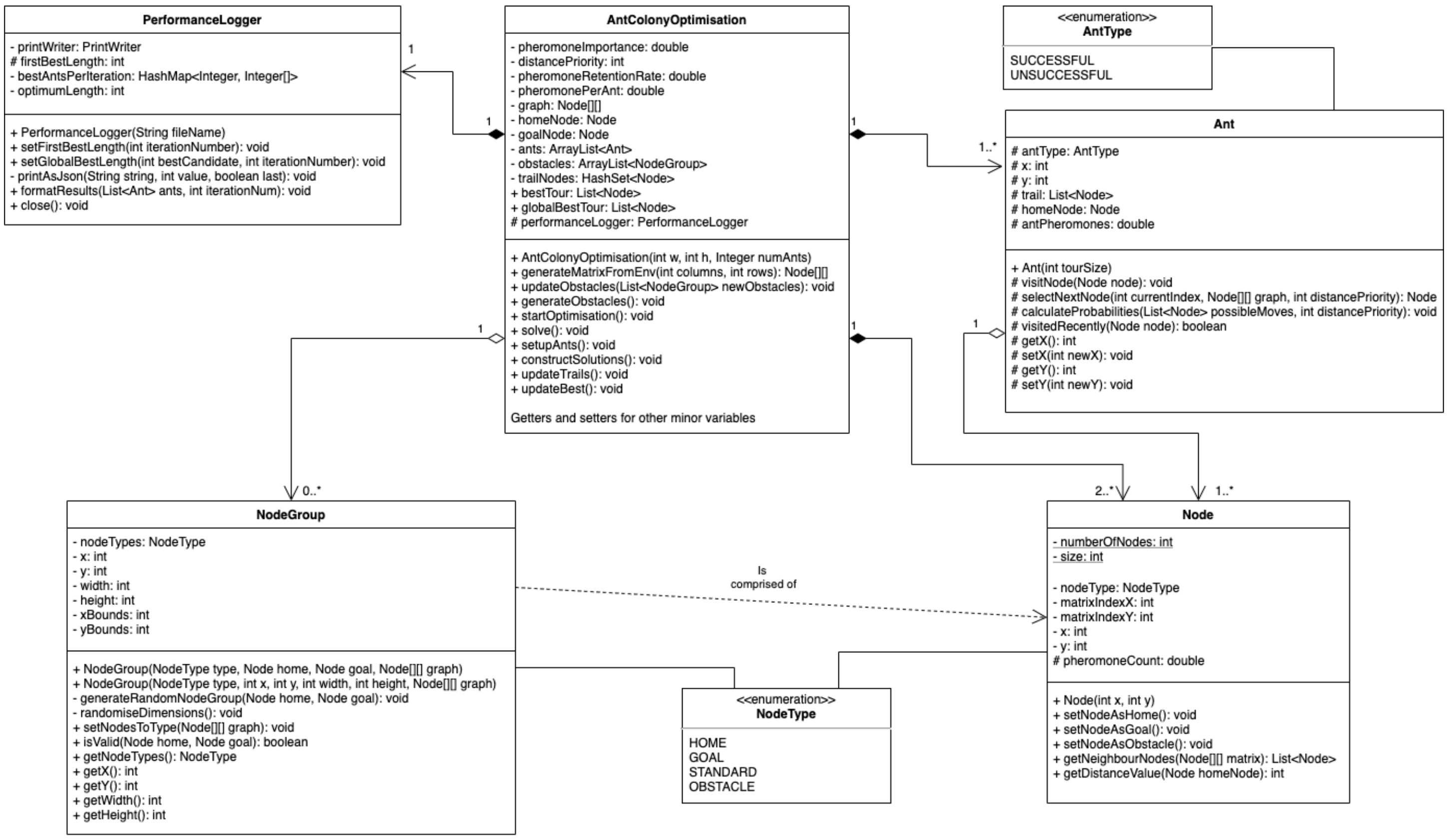
With these extensions to the design, the implementation of the project contains the following:

1. Ant Colony Optimisation Algorithm
2. Visual Display of the Algorithm
3. Simulation & Scenarios
4. Displaying Test Results using the Elastic Stack

4.1 Ant Colony Optimisation (ACO) Algorithm

The Ant Colony Optimisation (ACO) algorithm manages the movements of the ants on their journey from the home node, around the environment, and to the goal node. It manages their interactions with the environment, obstacles and scent trails, among many other things.

The composition of the ACO algorithm can be described using this class diagram:


PerformanceLogger
- printWriter: PrintWriter
# firstBestLength: int
- bestAntsPerIteration: HashMap<Integer, Integer[]>
- optimumLength: int
+ PerformanceLogger(String fileName)
+ setFirstBestLength(int iterationNumber): void
+ setGlobalBestLength(int bestCandidate, int iterationNumber): void
- printAsJson(String string, int value, boolean last): void
+ formatResults(List<Ant> ants, int iterationNum): void
+ close(): void
AntColonyOptimisation
- pheromoneImportance: double
- distancePriority: int
- pheromoneRetentionRate: double
- pheromonePerAnt: double
- graph: Node[][]
- homeNode: Node
- goalNode: Node
- ants: ArrayList<Ant>
- obstacles: ArrayList<NodeGroup>
- trailNodes: HashSet<Node>
+ bestTour: List<Node>
+ globalBestTour: List<Node>
# performanceLogger: PerformanceLogger
+ AntColonyOptimisation(int w, int h, Integer numAnts)
+ generateMatrixFromEnv(int columns, int rows): Node[][]
+ updateObstacles(List<NodeGroup> newObstacles): void
+ generateObstacles(): void
+ startOptimisation(): void
+ solve(): void
+ setupAnts(): void
+ constructSolutions(): void
+ updateTrails(): void
+ updateBest(): void
Getters and setters for other minor variables
<<enumeration>>
AntType
SUCCESSFUL
UNSUCCESSFUL
Ant
# antType: AntType
# x: int
# y: int
# trail: List<Node>
# homeNode: Node
# antPheromones: double
+ Ant(int tourSize)
# visitNode(Node node): void
# selectNextNode(int currentIndex, Node[][] graph, int distancePriority): Node
# calculateProbabilities(List<Node> possibleMoves, int distancePriority): void
# visitedRecently(Node node): boolean
# getX(): int
# setX(int newX): void
# getY(): int
# setY(int newY): void
1
1
1
1..*
1
1
1
1
0..*
2..*
1..*
NodeGroup
- nodeTypes: NodeType
- x: int
- y: int
- width: int
- height: int
- xBounds: int
- yBounds: int
+ NodeGroup(NodeType type, Node home, Node goal, Node[][] graph)
+ NodeGroup(NodeType type, int x, int y, int width, int height, Node[][] graph)
- generateRandomNodeGroup(Node home, Node goal): void
- randomiseDimensions(): void
+ setNodesToType(Node[][] graph): void
+ isValid(Node home, Node goal): boolean
+ getNodeTypes(): NodeType
+ getX(): int
+ getY(): int
+ getWidth(): int
+ getHeight(): int
Is
comprised of
<<enumeration>>
NodeType
HOME
GOAL
STANDARD
OBSTACLE
Node
- numberOfNodes: int
- size: int
- nodeType: NodeType
- matrixIndexX: int
- matrixIndexY: int
- x: int
- y: int
# pheromoneCount: double
+ Node(int x, int y)
+ setNodeAsHome(): void
+ setNodeAsGoal(): void
+ setNodeAsObstacle(): void
+ getNeighbourNodes(Node[][] matrix): List<Node>
+ getDistanceValue(Node homeNode): int

4.1.1 `AntColonyOptimisation` Class

The entirety of the path-finding algorithm is centred around the `AntColonyOptimisation` class, and this class controls and uses the other classes. The algorithm begins by calling `startOptimisation()` , but some initialisation takes place before that. The size, number of ants, home node, goal node and obstacles are all set either randomly or by the user. The environment is created using `generateMatrixFromEnv()` , obstacles are created using `updateObstacles()` , and ants are initialised using `setupAnts()` . The environment consists of a matrix of nodes defined by the `Node` class, and is explained in more detail below.

After this initialisation, the algorithm is ready to start by calling `startOptimisation()` , as mentioned above. The running of the program is a set of attempts, which are in turn comprised of a number of iterations. The `startOptimisation()` method sets up the environment and the ants for the beginning of each attempt, and contains a `for` loop containing the `solve()` method which runs each attempt.

The `solve()` method is responsible for running each of the iterations, and setting up the ants for the beginning of each iteration. For each iteration, the `constructSolutions()` method is run, and it utilises the classes `Ant` , `Node` and `NodeGroup` in order to find the shortest path from the home to the goal. As mentioned in section 3, the `constructSolutions()` method calculates the next position for each of the ants in the simulation using (i) the distance from the home node, and (ii) the pheromones present on the nodes surrounding the current node. This process continues until all the ants either find the goal, reach the maximum number of steps that they can take, or run out of nodes to visit.

Once all the ants have stopped and the `constructSolutions()` method has ended, the pheromones of the ants are dropped onto all the successful ants' trails. All ants have the same amount of pheromones that it can distribute along its trail, and these pheromones are divided evenly among the nodes in the ant's trail. This means that the shorter the trail, the more pheromones are present on that trail, and thus the shorter trail will be more favourable to ants in the next iteration than a longer trail.

The ants are then reset, and the ants run again on the updated environment containing the pheromones of the successful ants from the previous iteration.

Between each iteration, each node's pheromones dissipate somewhat. This means that the nodes which are travelled on less often eventually lose all of their pheromones. Only nodes that are travelled on often manage to retain their pheromones, and thus are the only ones that influence the ants over a number of iterations.

4.1.2 `Node` Class

The `Node` class is responsible for storing information of the environment. As mentioned above, the environment consists of a matrix of instances of the `Node` class. Each instance of `Node` contains information about the current state of that part of the environment, and this information is used to calculate the movements of the ants during the running of the ACO algorithm. Using the `NodeType` enumeration, instances of `Node` can take multiple forms or types, and are treated differently by the algorithm depending on their type. The available types are `HOME`, `GOAL`, `STANDARD` and `OBSTACLE`, and these different types influence how the ants move in the environment.

During the `constructSolutions()` method of the `AntColonyOptimisation` class, the ants need to know what their neighbour nodes are in order to choose their next move. The `getNeighbourNodes` method is called on the node that the ant is standing on, and it returns all nodes surrounding the current node that are not of type `OBSTACLE`. So in actuality, the ants do not "avoid" the obstacles in the environment, they just aren't given them as an option to move onto.

The ants also make use of the `getDistanceValue()` method in the `Node` class, which returns the distance from the current node to the home node using the Manhattan Distance formula. This distance value is then used by the ant to choose the next node to move to, with lower distance values being preferred over higher distance values. For a 7x7 matrix of nodes with the home node in the centre, the distance values of the nodes are:

43	44	45	46	45	44	43
44	45	46	47	46	45	44
45	46	47	48	47	46	45
46	47	48	Home	48	47	46
45	46	47	48	47	46	45
44	45	46	47	46	45	44
43	44	45	46	45	44	43

The `getDistanceValue()` method begins with the distance value being the number of nodes in the matrix (in this case it is a 7x7 matrix, so there are 49 nodes), and then it subtracts the Manhattan distance of the current node to the home node from the number of nodes in the matrix to give the distance value.

4.1.3 `NodeGroup` Class

Just like the `String` type is a collection of instances of `char` in Java, the `NodeGroup` is a collection of instances of `Node`. The exception is that each node in a `NodeGroup` is of the same type. This is how we implemented obstacles in our simulation, as they are a collection of instances of `Node` that are of the `NodeType` `OBSTACLE`.

Grouping instances of `Node` together into one class makes it easier to manage these nodes, as multiple instances of `NodeGroup` can exist, and all the nodes contained in a `NodeGroup` instance can be changed at once if an obstacle is deleted or added.

When designing the `NodeGroup` class, we decided to make it universal for all node types. This way, if we were to continue our work on this project, we could expand the functionality to contain `NodeGroup` instances of type `GOAL` or `HOME` to further investigate the abilities of our algorithm.

4.1.4 `Ant` Class

The `Ant` class is responsible for containing the information about each ant in the simulation and for calculating the ant's movements. Each instance of `Ant` represents a single ant, and contains information like the ant's position on the environment, how many pheromones the ant has to spread over the environment, and the nodes that the ant has visited (the ant's trail).

The ant's movements are calculated using the `selectNextNode()` method. As mentioned before, the selection of the next node is influenced by the ant's distance from the home node, and the pheromones present on the nodes that are surrounding the node the ant is standing on. There is also a small probability that the ant will choose a random node. This is to make sure new routes are found instead of ants being solely influenced by the pheromones and the distance from the home node.

If the probability of choosing a random node is not fulfilled, then `calculateProbabilities()` method is called within the `selectNextNode()` method. `calculateProbabilites()` returns a list of values for the surrounding nodes. These values are balanced between the ant's distance from the home node, and the number of pheromones present on the nodes they are referring to. Each of these values represents the probability of the ant choosing each respective node surrounding it. The node with the highest value associated with it after running the `calculateProbabilities()` method is chosen as the next node for the ant to visit.

Once a node is chosen using the `selectNextNode()` method, the ant moves to that node using the `visitNode()` method, which updates the ant's position.

Each ant, like the nodes, has a type. These types are implemented using the Enumeration class `AntType`, and contains the types `SUCCESSFUL` and `UNSUCCESSFUL`. Each ant starts with the type `UNSUCCESSFUL`, and once an ant reaches the goal node, it's type is changed from `UNSUCCESSFUL` to `SUCCESSFUL`. This type is used by the `AntColonyOptimisation` class to choose which ant trails to drop pheromones onto.

4.1.5 `PerformanceLogger` Class

The `PerformanceLogger` class was created to gather necessary test data while an instance of `AntColonyOptimisation` is running, format this data and send it to a results file in `res\testresults`. This class makes use of the Java IO `FileWriter` and `PrintWriter` classes.

An instance of `PerformanceLogger` is created upon each Attempt within an `AntColonyOptimisation`, and is passed a `filename` which is comprised of the timestamp of the Algorithm's initialisation and the current attempt number.

When initialised, the `PerfomanceLogger` creates a `FileWriter` with the corresponding `fileName` (as well as setting the `append` argument to `true`) and a `printWriter` taking the `fileWriter` as its argument.

The `PerformanceLogger` class has several methods dedicated to appending data to the `printWriter` at various stages in the ACO Algorithm lifecycle. Each one is passed a number of arguments from the `AntColonyOptimisation` class and arranges them into key: value pairs in a JSON format; designed to be read by the AWS Elasticsearch cluster we are using to aggregate our test data.

The main method included in `PerformanceLogger` is an overloaded method `formatResults()`, which allows different arguments to be passed at each stage in any attempt.

- The initialising `formatResults()` formats data upon initialisation of an `AntColonyOptimisation` instance, including the Attempt Number, Home node, Goal node and Obstacle Co-Ordinates, Pheromone Importance and Distance Priority. This also includes metadata for sending this data to elasticsearch
- During each iteration, `formatResults()` is called with data specific to each ant throughout an iteration, its `SUCCESSFUL` or `UNSUCCESSFUL` type, its trail length and its trail
- At the end of each iteration, `formatResults()` is called with an overview of all ants at the end of that iteration; the number of `SUCCESSFUL` or `UNSUCCESSFUL` ants, the iteration `bestTrail` and its length.
- Once all iterations within an attempt have taken place, `formatResults()` is called with attempt-wide data, including the overall `globalBestTour`, the iteration at which this global best was reached as well as the number of ants which found this globalBestTour.

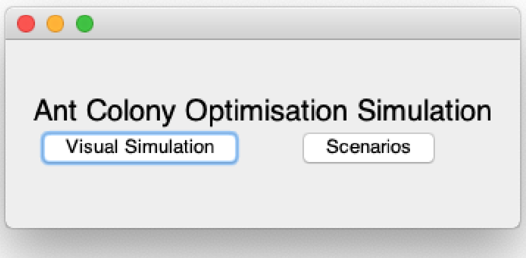
Once the attempt is complete, the `close()` method is called from `ÀntColonyOptimisation`, and the `printWriter` within the `performanceLogger` instance is closed, with all the data formatted in the newly created JSON results file.

The next two components of the project are accessed through the GUI, and allow the Ant Colony Optimisation algorithm to be run in two different ways:

1. Through a visual simulation where every calculation and movement of the algorithm is visualised.

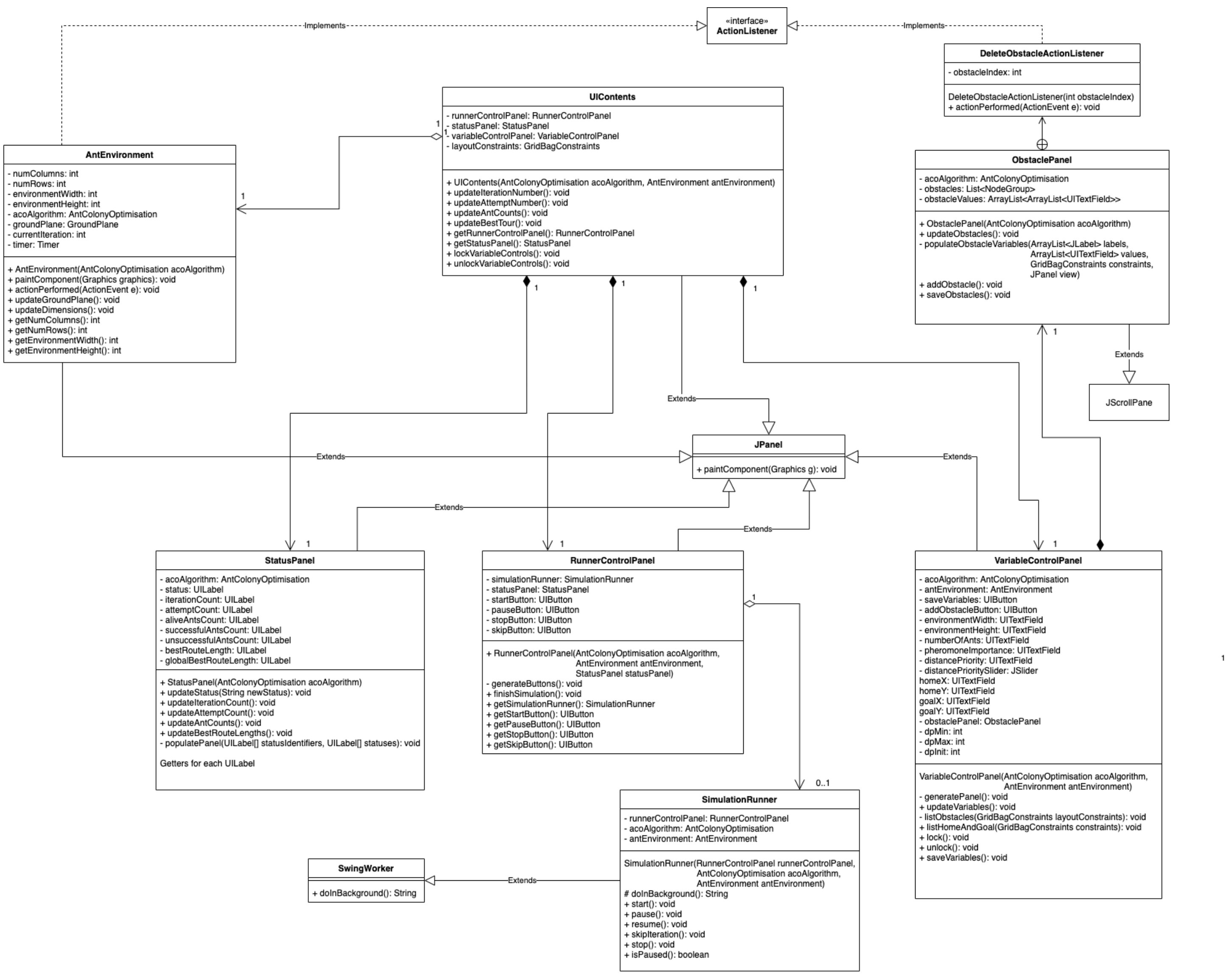
2. Through a scenario, where an environment is set by the user (which we call a "scenario"), and the algorithm is then run multiple times on the same environment, while systematically changing the algorithm variables to different combinations.

The user can choose between running a visual simulation, or a scenario in the "Simulation Options Window", which is the first window shown to the window, and is displayed below:

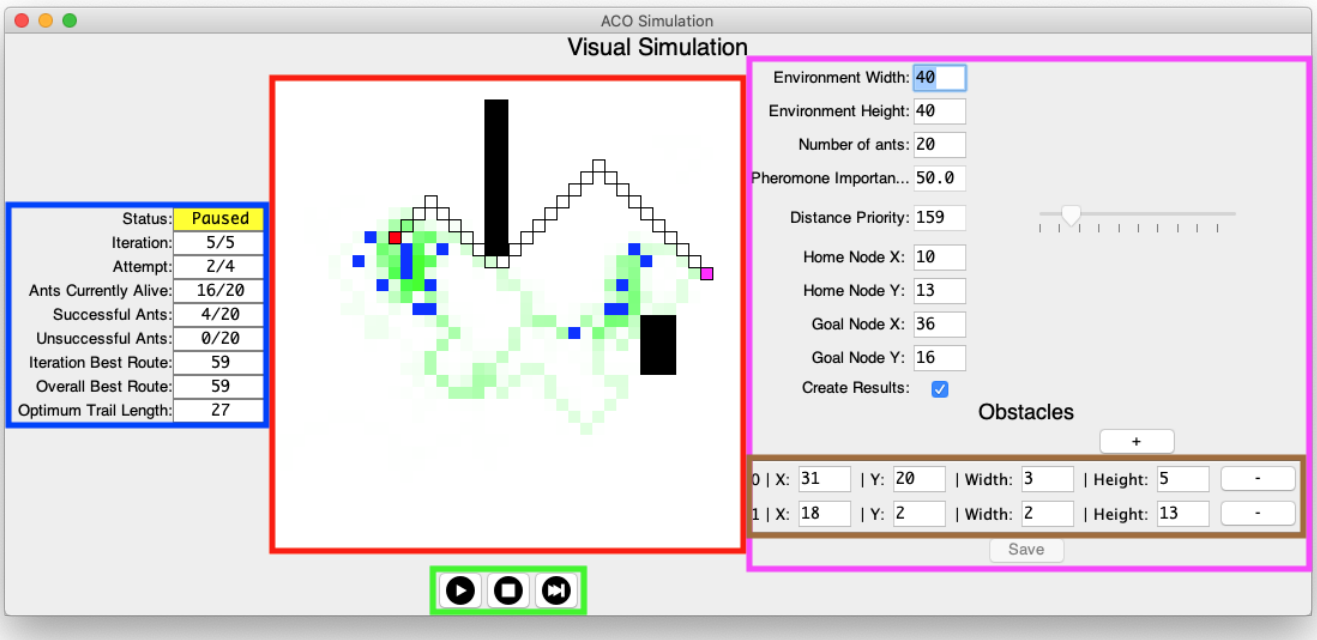


4.2 Visual Display of the Algorithm

If the user pressed the "Visual Simulation" button in the simulation options window, then the visual simulation GUI will be shown. The visual simulation GUI is represented by the following class diagram:


«interface»
ActionListener
Implements
Implements
DeleteObstacleActionListener
- obstacleIndex: int
DeleteObstacleActionListener(int obstacleIndex)
+ actionPerformed(ActionEvent e): void
UIContents
- runnerControlPanel: RunnerControlPanel
- statusPanel: StatusPanel
- variableControlPanel: VariableControlPanel
- layoutConstraints: GridBagConstraints
+ UIContents(AntColonyOptimisation acoAlgorithm, AntEnvironment antEnvironment)
+ updateIterationNumber(): void
+ updateAttemptNumber(): void
+ updateAntCounts(): void
+ updateBestTour(): void
+ getRunnerControlPanel(): RunnerControlPanel
+ getStatusPanel(): StatusPanel
+ lockVariableControls(): void
+ unlockVariableControls(): void
AntEnvironment
- numColumns: int
- numRows: int
- environmentWidth: int
- environmentHeight: int
- acoAlgorithm: AntColonyOptimisation
- groundPlane: GroundPlane
- currentIteration: int
- timer: Timer
+ AntEnvironment(AntColonyOptimisation acoAlgorithm)
+ paintComponent(Graphics graphics): void
+ actionPerformed(ActionEvent e): void
+ updateGroundPlane(): void
+ updateDimensions(): void
+ getNumColumns(): int
+ getNumRows(): int
+ getEnvironmentWidth(): int
+ getEnvironmentHeight(): int
ObstaclePanel
- acoAlgorithm: AntColonyOptimisation
- obstacles: List<NodeGroup>
- obstacleValues: ArrayList<ArrayList<UITextField>>
+ ObstaclePanel(AntColonyOptimisation acoAlgorithm)
+ updateObstacles(): void
- populateObstacleVariables(ArrayList<JLabel> labels, ArrayList<UITextField> values, GridBagConstraints constraints, JPanel view)
+ addObstacle(): void
+ saveObstacles(): void
Extends
JScrollPane
Extends
JPanel
+ paintComponent(Graphics g): void
Extends
Extends
Extends
Extends
StatusPanel
- acoAlgorithm: AntColonyOptimisation
- status: UILabel
- iterationCount: UILabel
- attemptCount: UILabel
- aliveAntsCount: UILabel
- successfulAntsCount: UILabel
- unsuccessfulAntsCount: UILabel
- bestRouteLength: UILabel
- globalBestRouteLength: UILabel
+ StatusPanel(AntColonyOptimisation acoAlgorithm)
+ updateStatus(String newStatus): void
+ updateIterationCount(): void
+ updateAttemptCount(): void
+ updateAntCounts(): void
+ updateBestRouteLengths(): void
- populatePanel(UILabel[] statusIdentifiers, UILabel[] statuses): void
Getters for each UILabel
RunnerControlPanel
- simulationRunner: SimulationRunner
- statusPanel: StatusPanel
- startButton: UIButton
- pauseButton: UIButton
- stopButton: UIButton
- skipButton: UIButton
+ RunnerControlPanel(AntColonyOptimisation acoAlgorithm, AntEnvironment antEnvironment, StatusPanel statusPanel)
- generateButtons(): void
+ finishSimulation(): void
+ getSimulationRunner(): SimulationRunner
+ getStartButton(): UIButton
+ getPauseButton(): UIButton
+ getStopButton(): UIButton
+ getSkipButton(): UIButton
VariableControlPanel
- acoAlgorithm: AntColonyOptimisation
- antEnvironment: AntEnvironment
- saveVariables: UIButton
- addObstacleButton: UIButton
- environmentWidth: UITextField
- environmentHeight: UITextField
- numberOfAnts: UITextField
- pheromoneImportance: UITextField
- distancePriority: UITextField
- distancePrioritySlider: JSlider
homeX: UITextField
homeY: UITextField
goalX: UITextField
goalY: UITextField
- obstaclePanel: ObstaclePanel
- dpMin: int
- dpMax: int
- dpInit: int
VariableControlPanel(AntColonyOptimisation acoAlgorithm, AntEnvironment antEnvironment)
- generatePanel(): void
+ updateVariables(): void
- listObstacles(GridBagConstraints layoutConstraints): void
+ listHomeAndGoal(GridBagConstraints constraints): void
+ lock(): void
+ unlock(): void
+ saveVariables(): void
SimulationRunner
- runnerControlPanel: RunnerControlPanel
- acoAlgorithm: AntColonyOptimisation
- antEnvironment: AntEnvironment
SimulationRunner(RunnerControlPanel runnerControlPanel, AntColonyOptimisation acoAlgorithm, AntEnvironment antEnvironment)
# doInBackground(): String
+ start(): void
+ pause(): void
+ resume(): void
+ skipIteration(): void
+ stop(): void
+ isPaused(): boolean
SwingWorker
+ doInBackground(): String
Extends
1
1
1
1
1
1
1
1
1
1
1
0..1
1

The layout of the visual simulation GUI is shown below, where the coloured boxes represent the different components found in the class diagram above.



Status:	Paused
Iteration:	5/5
Attempt:	2/4
Ants Currently Alive:	16/20
Successful Ants:	4/20
Unsuccessful Ants:	0/20
Iteration Best Route:	59
Overall Best Route:	59
Optimum Trail Length:	27

Environment Width:	40
Environment Height:	40
Number of ants:	20
Pheromone Importan...	50.0
Distance Priority:	159
Home Node X:	10
Home Node Y:	13
Goal Node X:	36
Goal Node Y:	16
Create Results:	✓

0 | X: 31 | Y: 20 | Width: 3 | Height: 5
1 | X: 18 | Y: 2 | Width: 2 | Height: 13

- Red Box: The `AntEnvironment` panel
- Green Box: The `RunnerControlPanel`
- Blue Box: The `StatusPanel`
- Pink Box: The `VariableControlPanel`
- Brown Box: The `ObstaclePanel`

All of these panels are linked together in one window, which is controlled by the `UIContents` class.

4.2.1 `AntEnvironment` panel

This panel displays the workings of the `AntColonyOptimisation` class visually. It consists of nodes and the ants on those nodes.

The Ants:

The ants in the environment are representative of instances of the `Ant` class, and are displayed on the environment as blue squares. They move around the given environment, with their movements dictated by the `AntColonyOptimisation` class. The `AntEnvironment` updates with each change to each ant's position. Once the ants reach the goal, they disappear from the visualisation as they have completed their task. If they run out of neighbours or run out of steps to take, their colour changes from blue to gray, and they stop moving.

The Nodes:

The nodes/squares in the environment that the ants move over are representative of instances of the `Node` class. The different types that a node can be are also represented on the environment in the following ways:

- `NodeType.HOME` : Represents the home, where the ants start their journey, and is visually denoted on the environment as a red square.
- `NodeType.GOAL` : Represents the goal, which the ants need to reach, and is visually denoted on the environment as a magenta square.
- `NodeType.OBSTACLE` : Represents obstacles on the environment, which the ants cannot pass through or stand on, and is visually denoted on the environment as a black square.
- `NodeType.STANDARD` : Represents a traversable area of the environment, which the ants use to reach the goal, and is visually denoted on the environment as a white square. Standard nodes also display their pheromone count visually by their shade of green. The more green a node is, the more pheromones are present on that node.
- The optimal trail: The optimal trail is shown as a series of black squares surrounding the nodes which make up the shortest route from the home node to the goal node. This optimal trail is calculated using the Breadth First Search algorithm, and is only one of many possible optimal routes. The ants aim to generate a trail that is of the same length as the optimal trail.

4.2.2 `RunnerControlPanel`

This panel allows the user to control the running of the simulation. Using this panel, users can:

- Start the simulation running.
- Stop the simulation.
- Pause the simulation.
- Resume the simulation when it's paused.
- Fast-forward/skip to the next iteration.

When the simulation is fast-forwarded, the ant colony optimisation algorithm still runs to the end of that iteration, it just removes the animation delay, so the algorithm can run at full speed.

4.2.3 `StatusPanel`

This panel contains all information on the current iteration of the ant colony optimisation algorithm, including:

- Simulation status.
- Iteration number.
- Attempt number.
- Number of ants currently alive in the simulation.
- Number of successful ants.
- Number of unsuccessful ants.
- The best route found in the current iteration.
- The overall best route found across all iterations and attempts.

4.2.4 `VariableControlPanel`

This panel allows the user to change the values of specific variables that affect the running of the algorithm. These variables are:

- Environment width.
- Environment height.
- Number of ants.
- Pheromone importance.
- Distance priority.
- Home node X and Y coordinates.
- Goal node X and Y coordinates.
- A checkbox to specify whether or not to gather results on this run of the algorithm.
- Number of obstacles, each obstacle's X and Y coordinates, width, and height.

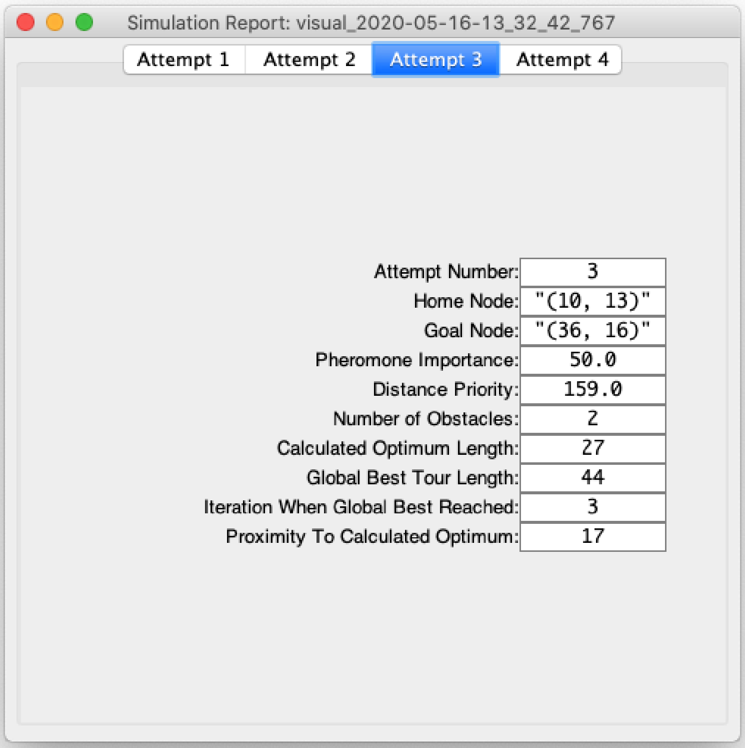
This panel allows different values of variables to be tested, and allows the user to see how each variable changes the working of the algorithm. It also helped us when fine-tuning the algorithm, and allowed us to construct test environments for extensive variable testing.

The user must click the "Save" button at the bottom of the `VariableControlPanel` in order for the changes they made to the variables to be applied in the next running of the ant colony optimisation algorithm.

4.2.5 `ObstaclePanel`

This panel is a sub-panel of the `VariableControlPanel` , and allows the user to specify specific coordinates of obstacles along with their widths and heights. When obstacles are added, they are assigned a random X, Y, width and height that are not overlapping the home and goal nodes. The user can then use the `ObstaclePanel` to assign new values to each added obstacle, or remove obstacles as they please.

If the user has checked the box "Create Results", and the simulation has been run to completion or has been stopped by the user, then the "Results Window" will be shown. Here, the user can see all the metrics gathered by the program during the running of the visual simulation. The GUI of the results window is shown below:



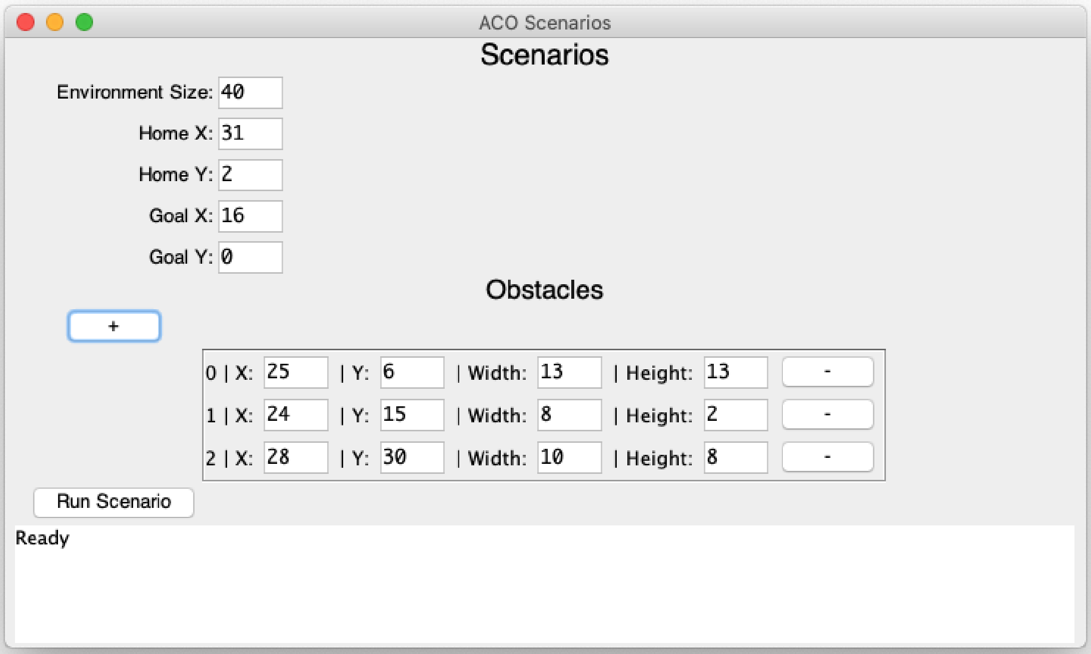
Attempt Number:	3
Home Node:	"(10, 13)"
Goal Node:	"(36, 16)"
Pheromone Importance:	50.0
Distance Priority:	159.0
Number of Obstacles:	2
Calculated Optimum Length:	27
Global Best Tour Length:	44
Iteration When Global Best Reached:	3
Proximity To Calculated Optimum:	17

4.3 Scenarios

Scenarios allow the user to specify a certain environment to run the algorithm on multiple times while systematically varying the environment variables.

4.3.1 Scenarios GUI

If the user pressed the "Scenarios" button in the simulation options window, then the scenarios GUI will be shown. A screenshot of the scenarios GUI is shown below:



Environment Size:	40
Home X:	31
Home Y:	2
Goal X:	16
Goal Y:	0

	X:	Y:	Width:	Height:
0	25	6	13	13
1	24	15	8	2
2	28	30	10	8

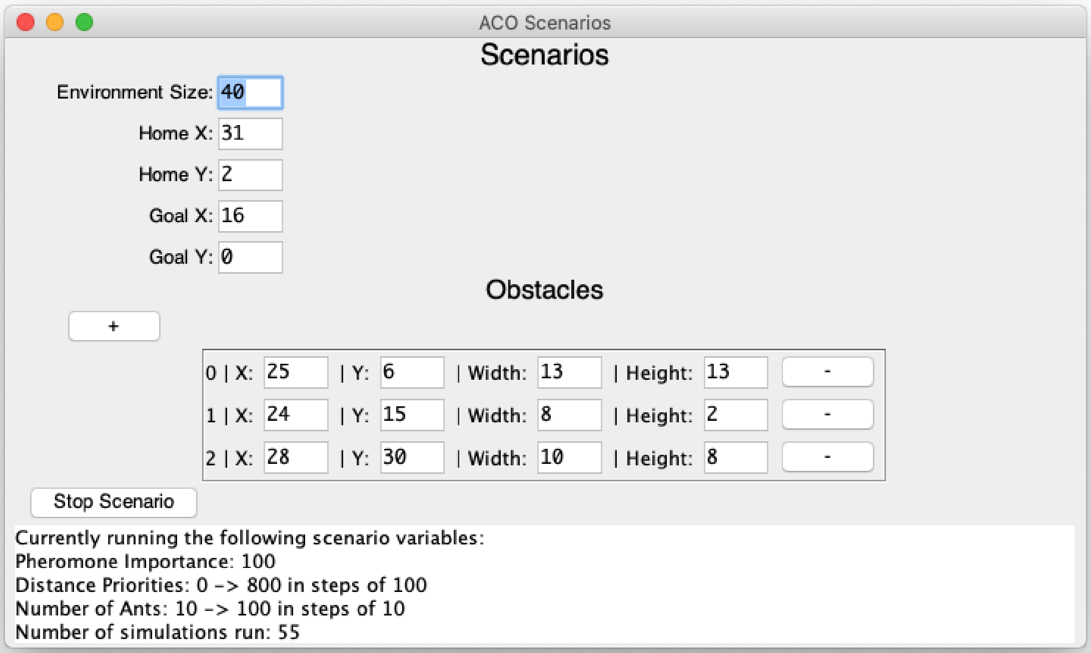
Ready

This window allows the user to specify an environment in which many iterations of the algorithm will be run. The user can specify the square environment size (rectangular environments are restricted for scenarios), the home coordinates, the goal coordinates, and the obstacles present in the environment. The same class is used for both the obstacles' specification box in the scenario window, and the corresponding obstacles' specification box in the visual simulation window (which is denoted by the brown box in the visual simulation GUI screenshot).

Once the "Run Scenario" button is clicked, the window changes to show the status of the current scenario, specifying which range of variable values are currently being tested on the specified environment. The variable values that are changed for each run are:

1. Pheromone Importance: This value changes in increments of 100, starting at 0 and up to a maximum of 400.
2. Distance Priority: This value changes in increments of 100, starting at 0 and up to a maximum of 800.
3. Number of Ants: This value changes in increments of 20, starting at 10 and up to a maximum of 100.

A screenshot of the scenarios GUI while the scenario is running is shown below:



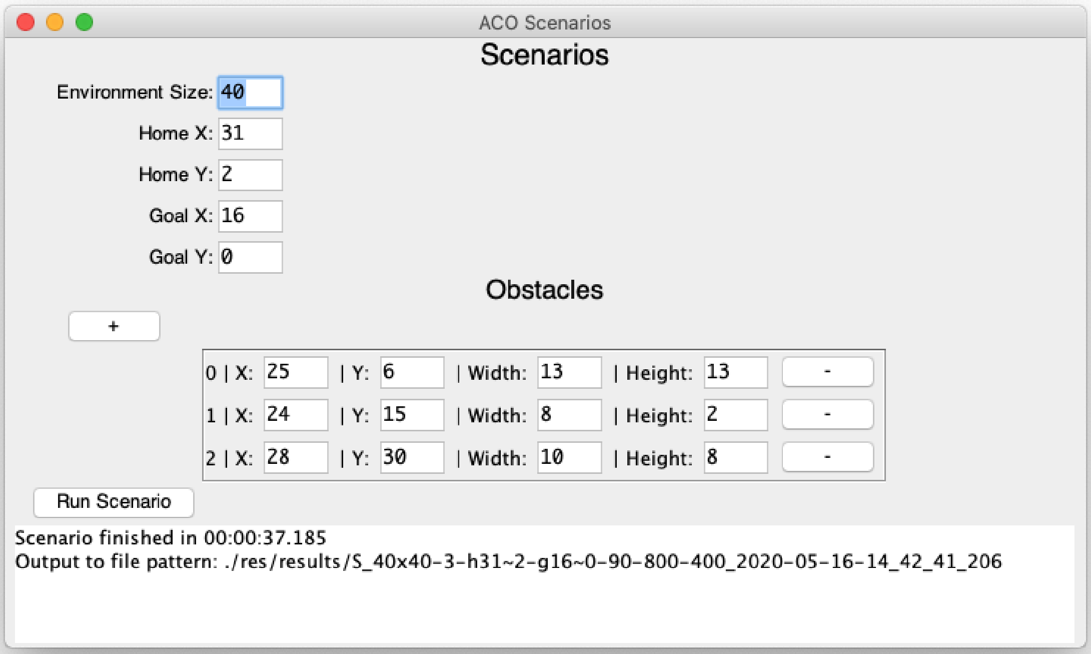
Environment Size: 40
Home X: 31
Home Y: 2
Goal X: 16
Goal Y: 0

0 | X: 25 | Y: 6 | Width: 13 | Height: 13
1 | X: 24 | Y: 15 | Width: 8 | Height: 2
2 | X: 28 | Y: 30 | Width: 10 | Height: 8

```
Currently running the following scenario variables:
Pheromone Importance: 100
Distance Priorities: 0 -> 800 in steps of 100
Number of Ants: 10 -> 100 in steps of 10
Number of simulations run: 55
```

Once the scenario runs until completion, or if the user presses the "Stop Scenario" button, then the results gathered are written to the "res/results/" directory in the project directory. The runtime of the scenario is displayed to the user, and the file location and file pattern of the results file are also shown.

A screenshot of the scenarios GUI when the scenario has finished running is shown below:



Environment Size: 40
Home X: 31
Home Y: 2
Goal X: 16
Goal Y: 0

0 | X: 25	| Y: 6	| Width: 13	| Height: 13	-
1 | X: 24	| Y: 15	| Width: 8	| Height: 2	-
2 | X: 28	| Y: 30	| Width: 10	| Height: 8	-

```
Scenario finished in 00:00:37.185
Output to file pattern: ./res/results/S_40x40-3-h31~2-g16~0-90-800-400_2020-05-16-14_42_41_206
```

4.3.2 `Scenario` Class

This class is responsible for the testing of the entire `AntColonyOptimsation` and allows extensive running of the algorithm across a given environment, with the environment variables varying between each running of the algorithm. This allows us to gather extensive metrics to examine how the algorithm runs as a whole, and to find out the optimum values of the variables that influence the running of the algorithm.

For each instance of the `Scenario` class, the `runScenario()` method runs the given environment 225 times while varying the pheromone importance, distance priority and number of ants for each run. The results from each run is collected, and each run's results is stored in its own results file in json format.

4.4 Displaying Test Results using the Elastic Stack

We made use of the Elastic Stack for aggregating, displaying and analysing the data gathered while testing our algorithm, in particular the following tools;

- An AWS Elasticsearch cluster to contain, search through and analyse our data
- Kibana plugin to organise and visualise our data
- Filebeat to automate the uploading of our results file to the AWS Elasticsearch cluster

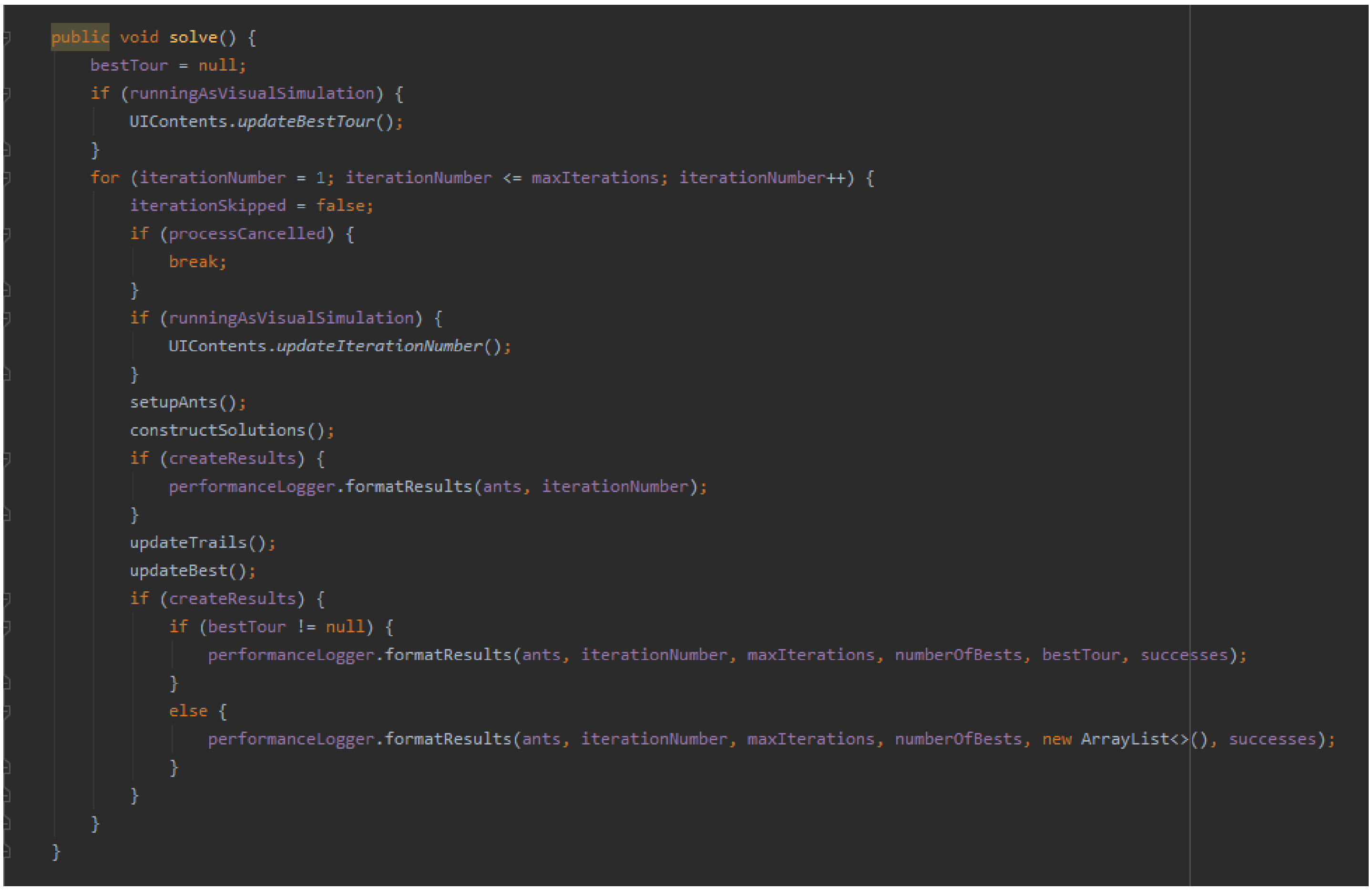
For more information on the elastic stack, visit their documentation here

Our process for working with this data was as follows:

- Create a results file for every attempt using the `PerformanceLogger` class.
- Configure a locally installed filebeat service to harvest the data from our `./res/results` folder. This would periodically scan this folder and look for any new JSON files present to send to our AWS Elasticsearch cluster.
- Arrange and display the results visually in a Kibana dashboard, a plugin allowing us to analyse the data gathered against each other in a concise way.

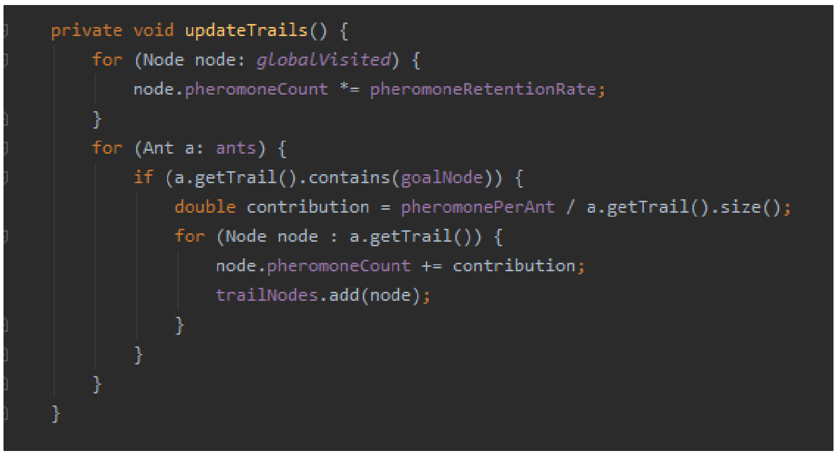
5. Sample Code

The `solve` method in `AntColonyOptimisation`, which runs the algorithm once initiated.



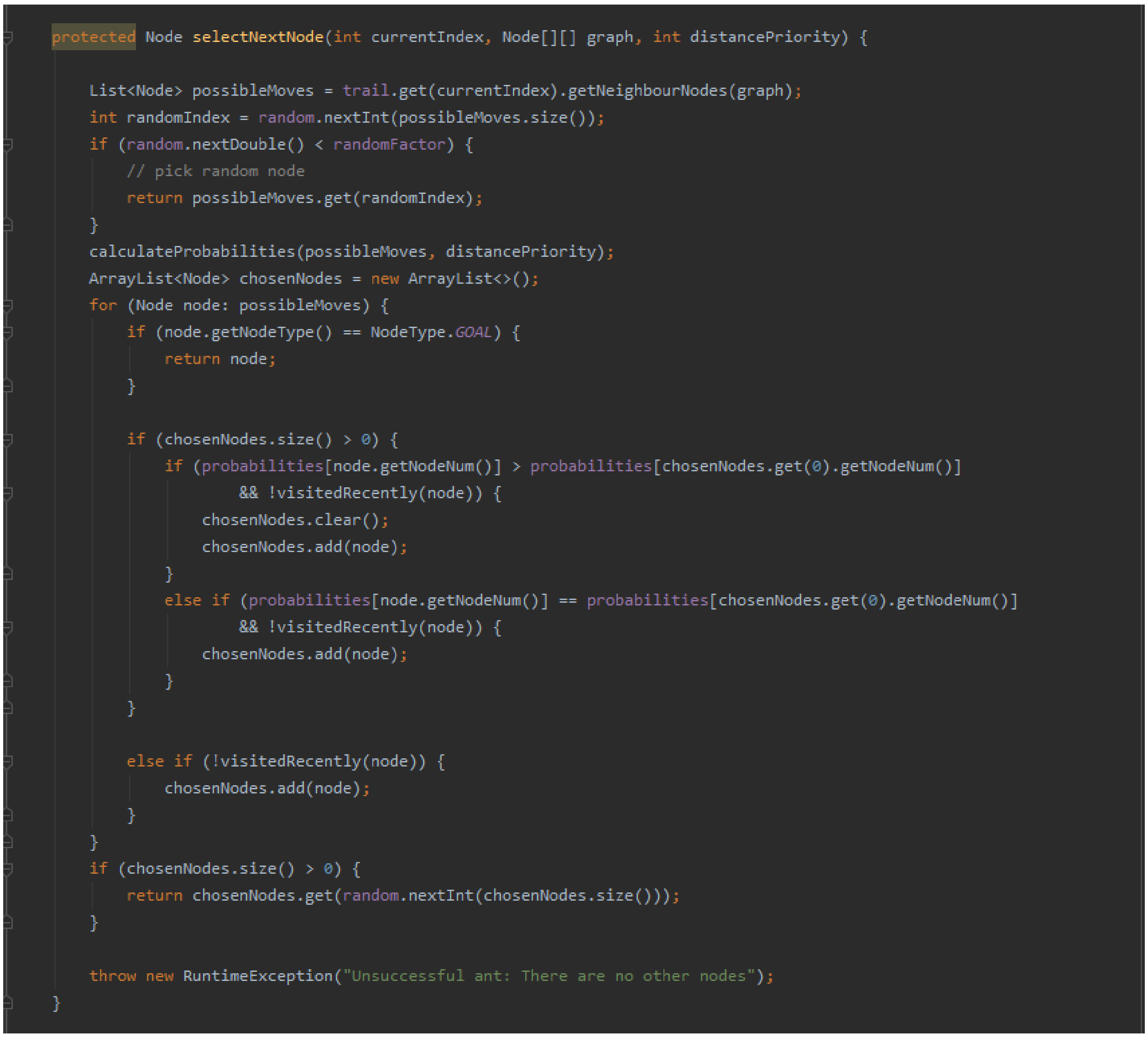
```
public void solve() {
    bestTour = null;
    if (runningAsVisualSimulation) {
        UIContents.updateBestTour();
    }
    for (iterationNumber = 1; iterationNumber <= maxIterations; iterationNumber++) {
        iterationSkipped = false;
        if (processCancelled) {
            break;
        }
        if (runningAsVisualSimulation) {
            UIContents.updateIterationNumber();
        }
        setupAnts();
        constructSolutions();
        if (createResults) {
            performanceLogger.formatResults(ants, iterationNumber);
        }
        updateTrails();
        updateBest();
        if (createResults) {
            if (bestTour != null) {
                performanceLogger.formatResults(ants, iterationNumber, maxIterations, numberOfBests, bestTour, successes);
            }
            else {
                performanceLogger.formatResults(ants, iterationNumber, maxIterations, numberOfBests, new ArrayList<>(), successes);
            }
        }
    }
}
```

The `updateTrails` method in `AntColonyOptimisation` , which applies pheromone evaporation and checks for nodes used in ant paths to give a pheromone boost



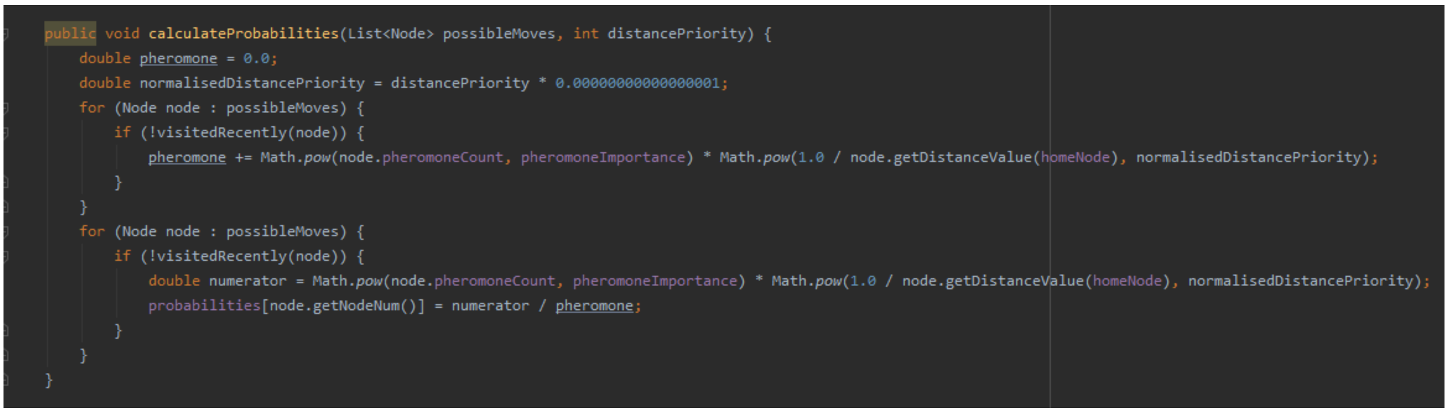
```
private void updateTrails() {
    for (Node node: globalVisited) {
        node.pheromoneCount *= pheromoneRetentionRate;
    }
    for (Ant a: ants) {
        if (a.getTrail().contains(goalNode)) {
            double contribution = pheromonePerAnt / a.getTrail().size();
            for (Node node : a.getTrail()) {
                node.pheromoneCount += contribution;
                trailNodes.add(node);
            }
        }
    }
}
```

The `selectNextNode` method in `Ant` , which picks the next node for an ant to visit in the matrix



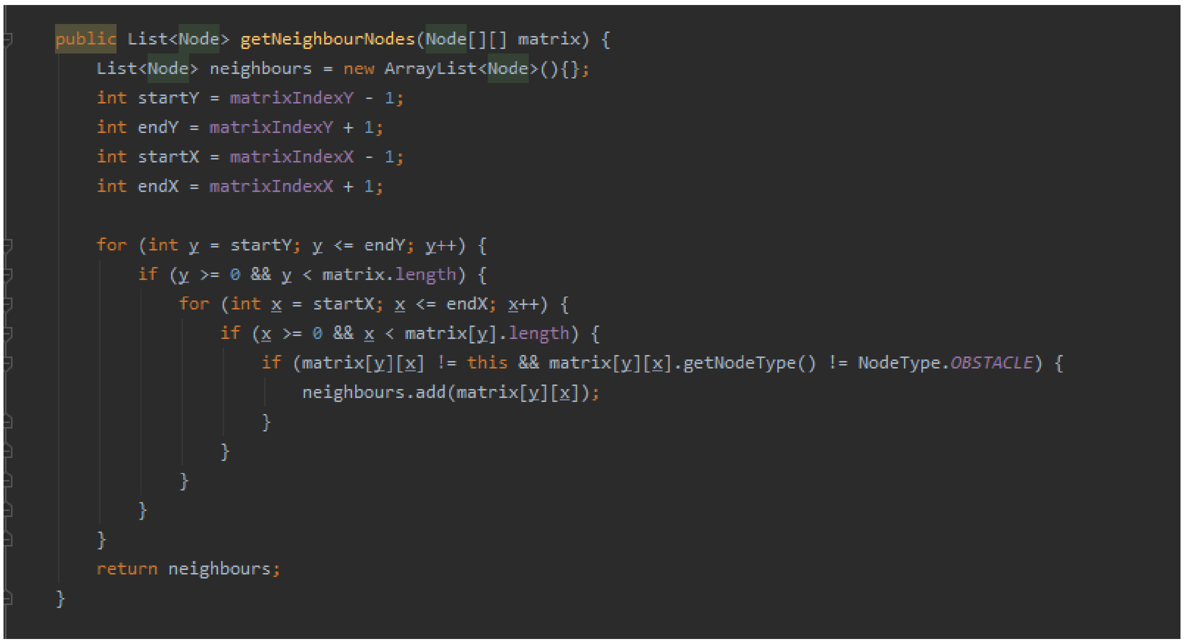
```java
protected Node selectNextNode(int currentIndex, Node[][] graph, int distancePriority) {

    List<Node> possibleMoves = trail.get(currentIndex).getNeighbourNodes(graph);
    int randomIndex = random.nextInt(possibleMoves.size());
    if (random.nextDouble() < randomFactor) {
        // pick random node
        return possibleMoves.get(randomIndex);
    }
    calculateProbabilities(possibleMoves, distancePriority);
    ArrayList<Node> chosenNodes = new ArrayList<>();
    for (Node node: possibleMoves) {
        if (node.getNodeType() == NodeType.GOAL) {
            return node;
        }

        if (chosenNodes.size() > 0) {
            if (probabilities[node.getNodeNum()] > probabilities[chosenNodes.get(0).getNodeNum()]
                    && !visitedRecently(node)) {
                chosenNodes.clear();
                chosenNodes.add(node);
            }
            else if (probabilities[node.getNodeNum()] == probabilities[chosenNodes.get(0).getNodeNum()]
                    && !visitedRecently(node)) {
                chosenNodes.add(node);
            }
        }

        else if (!visitedRecently(node)) {
            chosenNodes.add(node);
        }
    }
    if (chosenNodes.size() > 0) {
        return chosenNodes.get(random.nextInt(chosenNodes.size()));
    }

    throw new RuntimeException("Unsuccessful ant: There are no other nodes");
}
```

The `calculateProbabilities` method in `Ant` , which helps the select next node process



```java
public void calculateProbabilities(List<Node> possibleMoves, int distancePriority) {
    double pheromone = 0.0;
    double normalisedDistancePriority = distancePriority * 0.00000000000000001;
    for (Node node : possibleMoves) {
        if (!visitedRecently(node)) {
            pheromone += Math.pow(node.pheromoneCount, pheromoneImportance) * Math.pow(1.0 / node.getDistanceValue(homeNode), normalisedDistancePriority);
        }
    }
    for (Node node : possibleMoves) {
        if (!visitedRecently(node)) {
            double numerator = Math.pow(node.pheromoneCount, pheromoneImportance) * Math.pow(1.0 / node.getDistanceValue(homeNode), normalisedDistancePriority);
            probabilities[node.getNodeNum()] = numerator / pheromone;
        }
    }
}
```

The `getNeighbourNodes` method in `Node` , which fetches all neighbouring nodes that are visitable



```java
public List<Node> getNeighbourNodes(Node[][] matrix) {
    List<Node> neighbours = new ArrayList<Node>(){};
    int startY = matrixIndexY - 1;
    int endY = matrixIndexY + 1;
    int startX = matrixIndexX - 1;
    int endX = matrixIndexX + 1;

    for (int y = startY; y <= endY; y++) {
        if (y >= 0 && y < matrix.length) {
            for (int x = startX; x <= endX; x++) {
                if (x >= 0 && x < matrix[y].length) {
                    if (matrix[y][x] != this && matrix[y][x].getNodeType() != NodeType.OBSTACLE) {
                        neighbours.add(matrix[y][x]);
                    }
                }
            }
        }
    }
    return neighbours;
}
```

6. Problems Solved

6.1 Adapting to nature of problem

Using this ACO algorithm to find the minimum distance between two specified nodes within an open matrix instead of exhaustively traversing a fully connected matrix challenged us to rethink how the algorithm operated without changing the fundamental principles. This also required us to think about the differences between our problem and the traditional ACO problem while researching previous ACO algorithm implementations.

We had to take the following into consideration:

Defining a successful solution

Originally, the solutions constructed within the for a combinatorial problem such as the Travelling Salesman Problem, the goal of each ant was to create a route visiting every node within the construction matrix and then returning to the node at which it started. This is an exhaustive traversal; all solutions are valid and the combined length of each move is used to determine the shortest path.

The ants in this project are not required to visit every node in their graph, rather they all begin at the same node and try to find the same goal node within the graph in as few moves as possible. They are not required to visit every node in their environment, and are considered successful as soon as their trail contains the goal node. Each node is equidistant from all its neighbouring nodes and so the trail with the smallest number of nodes is considered the shortest.

Introducing an unsuccessful solution

Following on from our new definition of the successful trail, there now exists a case where an ant runs out of available moves before reaching the goal. In this case, when an ants trail does not contain the goal node, that ant will be considered unsuccessful and it will not apply positive pheromone update to any of the nodes in its trail.

Distance Heuristic

In the traditional ACO algorithm, each move made by an ant is determined probabilistically taking into account the pheromone value associated with each candidate node and its distance heuristic; favouring moves closer to the end of the route (i.e the final move, and the "goal" in TSP). As the problem we are trying to solve is trying to find a route to a goal node not known by the ants at the start, the use of the original distance heuristic had to be changed so that it did not rely on the position of the goal node. When addressing this change, we included a distance heuristic favouring nodes further from the home node rather than closer to the goal node, to encourage further exploration of the environment and minimise ants backtracking.

6.2 Determining the Optimum Solution

In order to accurately test for the fitness of this algorithm, we needed to determine the optimum solution of any given environment to compare results to.

We made use of a breadth-first-search traversal that searched the environment exhaustively until the goal node was found. This was implemented as a tree which took the home as the root node and, for each node visited, each of its unvisited neighbours would be included as that current node's children. Once found, the height of the tree between the goal node to the root home node would be taken as the optimum solution.

6.3 Animation

When we began the project, we had little-to-no prior knowledge of animation, so we needed to learn the basics of animation, and the basics of animation in Java. We also had no prior knowledge of Java Swing or Java AWT, which are the Java windowing tools.

In order to overcome this, we took various online tutorials on animation and Java Swing and AWT, and read the documentation on Java Swing and Java AWT. Our knowledge of Java Swing and AWT grew through trial and error, and by the end of this project, we have acquired a good understanding of both animation, and the ways to implement it in Java.

Another challenge with animation was ensuring that the UI could properly represent the workings of the Ant Colony Optimisation algorithm. This challenge was overcome by representing the environment as a matrix of nodes, and displaying each node as a uniform size in the UI.

6.4 Structure of UI

The way to structure the UI, and the way to transfer data from the Ant Colony Optimisation algorithm to the UI and vice-versa, were unknown to us in the beginning. In order for the simulation to be displayed, all ant positions, obstacle positions, goal and home positions needed to be known to the user interface.

We accomplished this by ensuring that the project was separated into two components; the Ant Colony Optimisation algorithm and the UI. This way, changes could be made to the UI without affecting the Ant Colony Optimisation algorithm, and both could be scaled independently.

6.5 Setting up the Elastic Stack

While aggregating all the results files within the project, we faced many challenges in setting up an AWS ElasticSearch cluster and harvesting the results folder automatically using Filebeat.

Elasticsearch requires JSON objects to parse and display, but require an index configuration object and an entire JSON object on a single line in order to be recognised by the cluster. In contrast, Filebeat expects to receive text based logfiles that don't automatically recognise and parse JSON objects. We had to configure filebeat to both recognise and extract JSON objects within any result file so that numbers, strings and objects could be treated separately when displaying the data in Kibana.

7. Results

To evaluate the fitness of our algorithm, we used three different `Scenario` types; a 'Predefined' scenario with a specified environment, a 'Random' Scenario which was generated at random, and a 'SpeedTest' Scenario which used the specified environments but utilised more iterations within attempts to further assess the number of iterations in which a solution was found.

The three factors we wished to analyse in our evaluation were the following:

- `Pheromone Importance` - whether a smaller or greater preference to nodes with a strong pheromone value would work more efficiently
- `Distance Priority` - whether a smaller or greater preference to nodes further away from the home would work more efficiently.
- `Number of Ants` - whether a smaller or larger colony would work more efficiently.

As we were measuring three separate factors; we had to ensure each possible combination was tested to find the best cross-section. To do this, each scenario comprised of a single environment in which many attempts were run. Each attempt used a different combination of these values until every possibility within our defined constraints was tried.

When measuring "efficiency", we focused on two areas: Accuracy and Speed. The accuracy of an attempt was a measure of how close to the precalculated optimum distance the algorithm would achieve by the end of its run. We logged the difference between the `globalBestTrail` and `precalculatedOptimumRoute` lengths, and favoured those nearing 0 as more accurate attempts. The speed of an attempt considered the iteration at which the best trail or optimum was reached. The less iterations it took an algorithm attempt to reach its `globalBestTrail` , the faster it would operate.

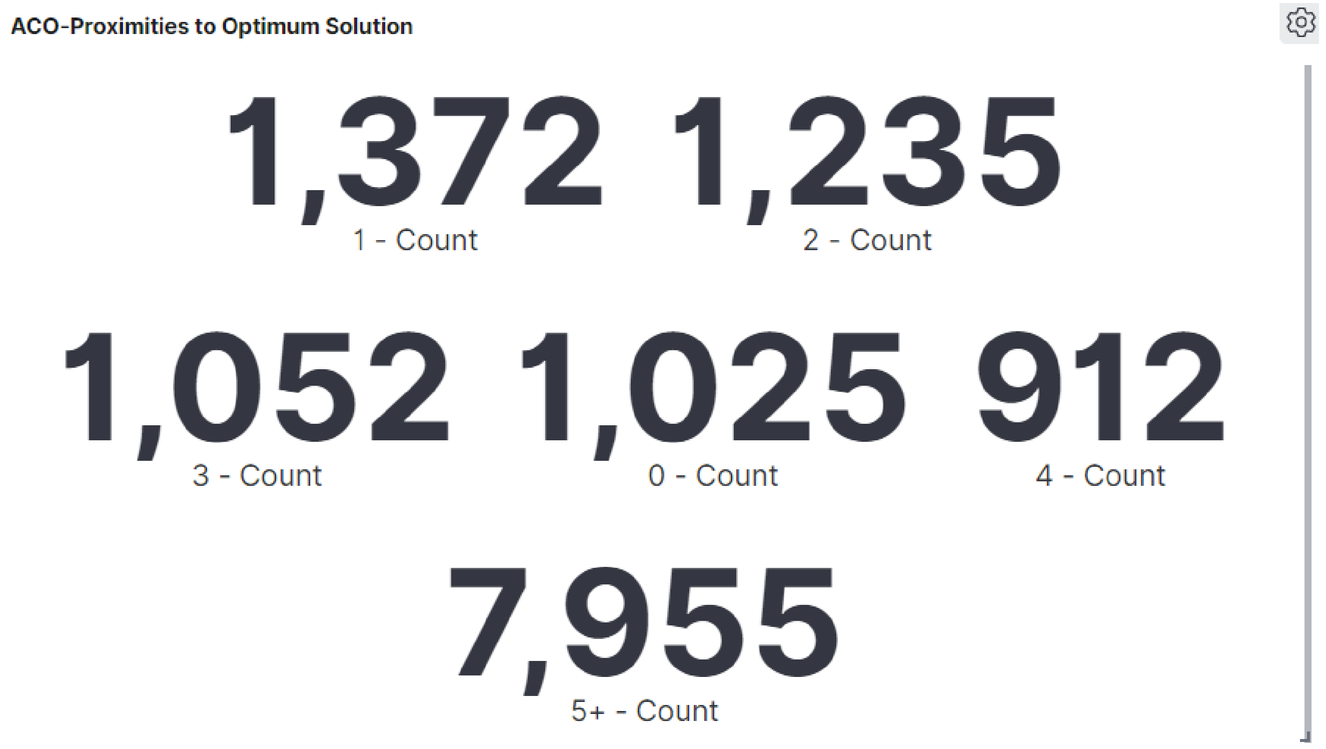
All results gathered throughout the testing process have been saved to an AWS Elasticsearch cluster and were analysed using the Kibana plugin.

7.1 Accuracy Statistics

The overall stats are as follows:

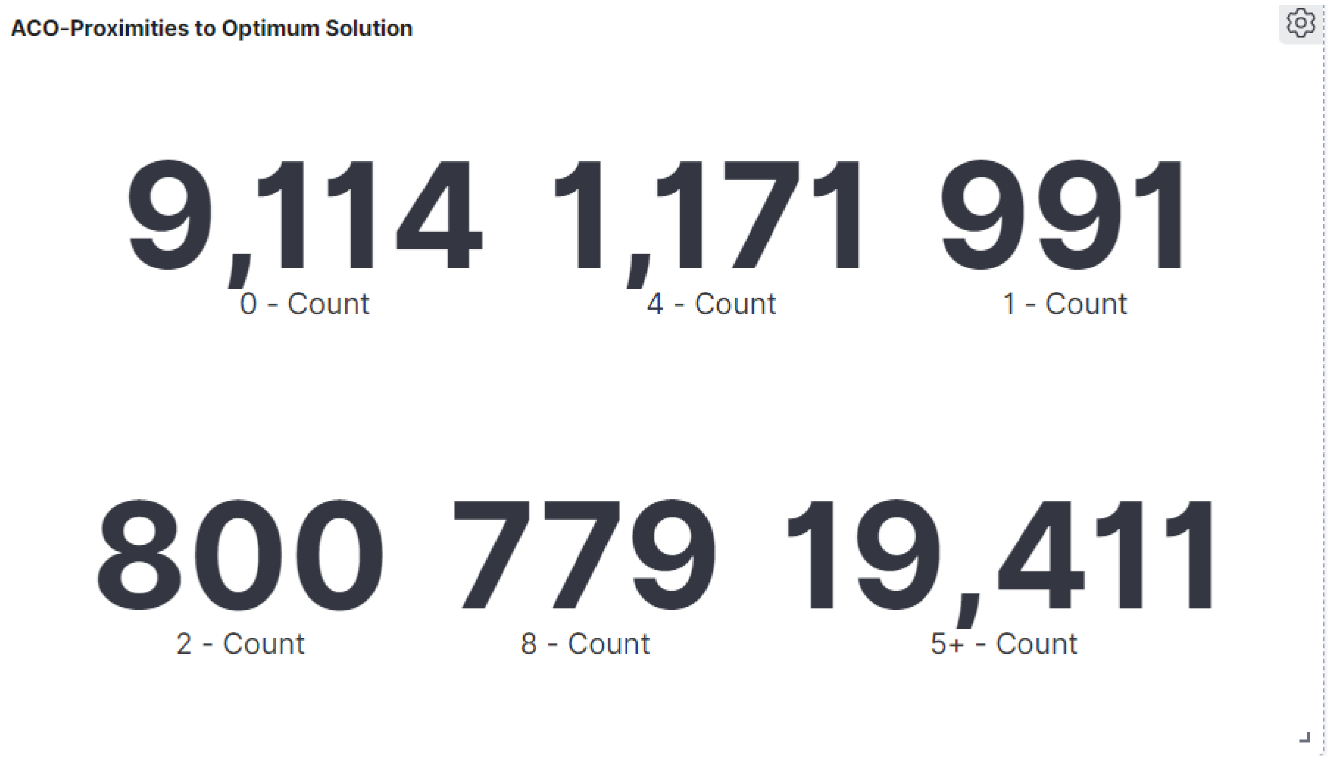
Out of 13,551 Predefined Scenario attempts

- 1,025 were fully accurate i.e managed to find the precalculated optimum (7.56%)
- 1,372 had best route lengths that were 1 node longer than the precalculated optimum (10.12%)
- 1,235 had best route lengths 2 nodes longer than the precalculated optimum (9.11%)
- The average proximity to the precalculated optimum was 8.839 nodes.



Out of 32,266 Random Scenario attempts

- 9,114 were fully accurate (28.25%)
- 991 had best routes that were 1 node longer than the precalculated optimum (3.07%)
- 800 had best routes that were 2 nodes longer than the precalculated optimum (2.48%)
- The average proximity to the precalculated optimum was 23.609 nodes.


ACO-Proximities to Optimum Solution

9,114 0 - Count
1,171 4 - Count
991 1 - Count
800 2 - Count
779 8 - Count
19,411 5+ - Count

7.2 Speed Test

A separate Speed centric test was run with an extended number of iterations to measure the number of iterations

For the 3,301 completely accurate attempts:

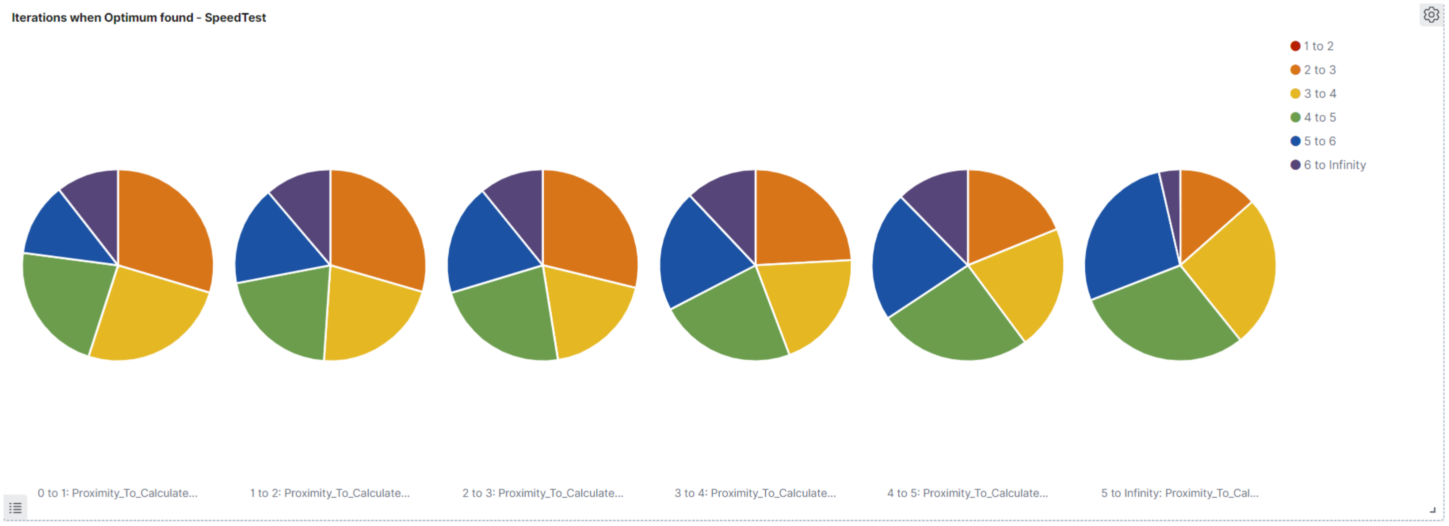
- 1,409 attempts found the optimum route in 2 moves (42.68%)
- 841 attempts found the optimum route in 3 moves (25.48%)
- 576 attempts found the optimum in 4 moves (17.45%)
- 423 attempts found the optimum in 5 moves (12.81%)
- 52 attempts found the optimum in over 5 moves (1.58%)

For the 2,362 attempts in which the best Trail length was 1 node off the optimum:

- 753 attempts found the optimum route in 2 moves (31.88%)
- 527 attempts found the optimum route in 3 moves (22.31%)
- 542 attempts found the optimum in 4 moves (22.95%)
- 472 attempts found the optimum in 5 moves (19.98%)
- 68 attempts found the optimum in over 5 moves (2.88%)

For the 2,362 attempts in which the best Trail length was 2 node off the optimum:

- 635 attempts found the optimum route in 2 moves (26.84%)
- 546 attempts found the optimum in 3 moves (23.08%)
- 577 attempts found the optimum route in 3 moves (24.39%)
- 544 attempts found the optimum in 5 moves (22.99%)
- 64 attempts found the optimum in over 5 moves (2.7%)

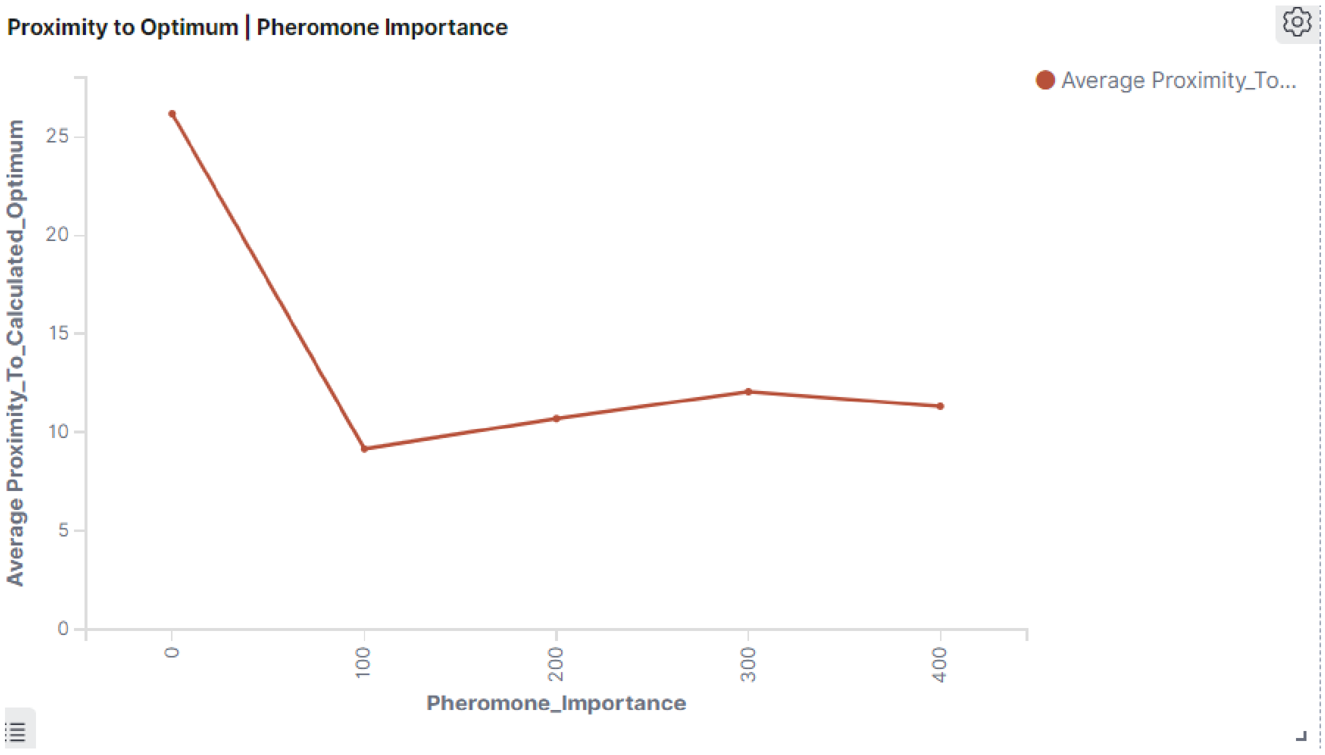


Based on this information, the algorithm would tend to reach it's own best length in 1-2 iterations, especially if that best length was closer to the precalculated optimum. There seems to be a slight correlation between an attempt taking longer to reach its own best length and being further from achieiving the precalculated optimum in general.

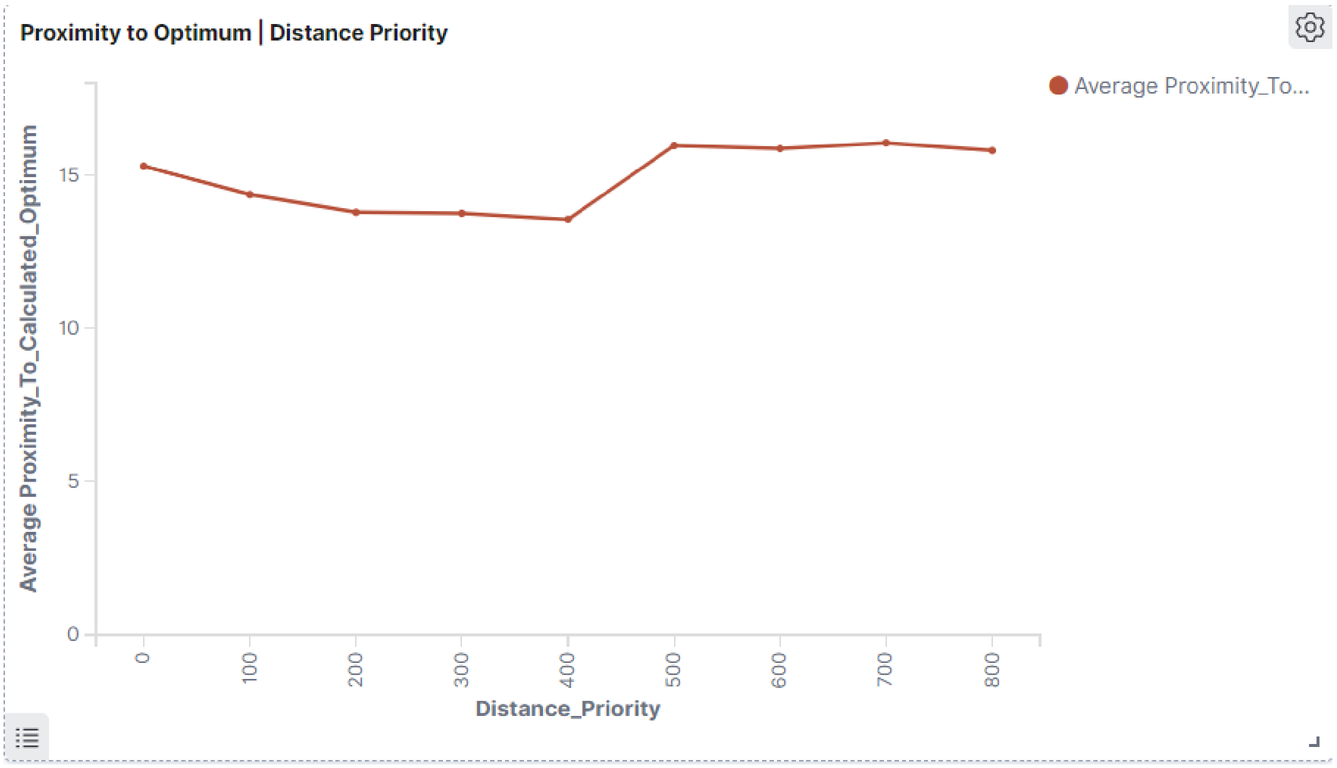
7.3 Cross-Section of Parameters

While investigating the best cross section of values between the pheromone importance, distance priority and number of ants, we generated the following graphs. Each is based off all Scenario types i.e for all 50,817 recorded attempts.

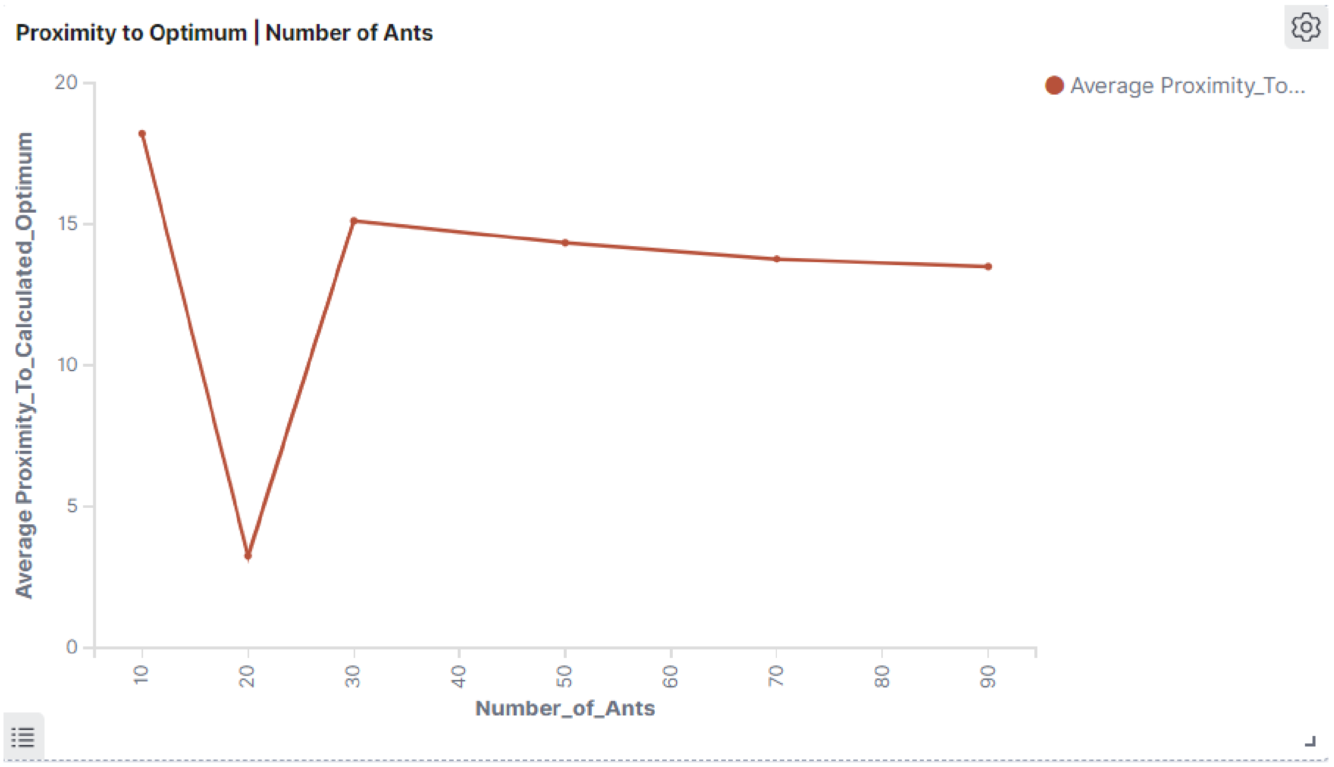
Average Accuracy with Pheromone Importance



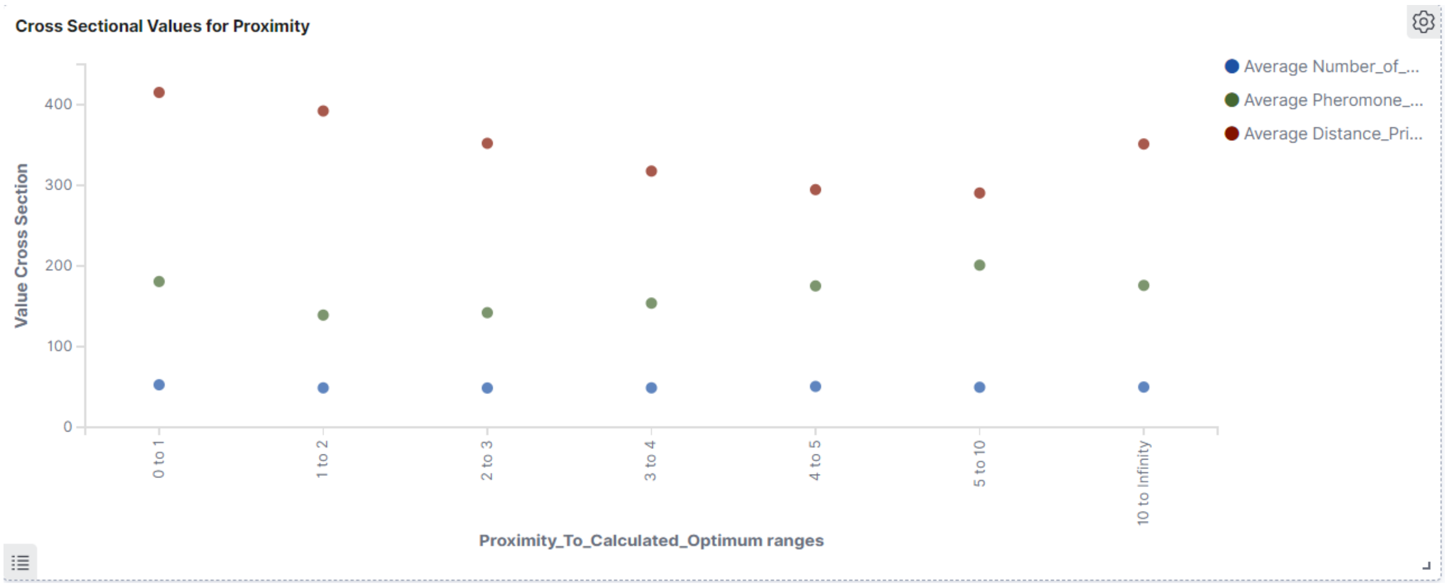
Average Accuracy with Distance Priority



Average Accuracy with Number of Ants



Cross section of values



This has the proximity to calculated optimum on the x-axis, with each of the averages of the different factor values represented as dots on the y axis. Red dot denotes the distance priority, green represents the pheromone importance and blue represents the number of ants. A lower proximity indicates a more accurate attempt, and so from this cross section we can gather the following 'ideal' parameters:

- Distance priority: 415.188
- Pheromone importanec: 180.741
- Number of ants: 53

8. Future Work

Having undertaken this project, learning about the Ant Colony Optimisation Algorithm and it's ability in a non-exhaustive problem context, there are a few directions in which this work can be taken forward; both in future research and real world application.

8.1 Further Research

The work undertaken here aimed to introduce the ant colony optimisation algorithm to a new problem context - an open plain construction matrix with a single goal to be found rather than a fully connected matrix where each must be visited. This greatly expands the capabilities of the algorithm.

However, for the purpose and scope of this project, the focus was primarily on the use of pheromone trails left by single ants and their impact on the colony as a whole over time. Impacts of the number of ants in a virtual colony and the influence of distance on the success rate of the optimisation algorithm were also studied here. Following on from the work in this project, more unexplored features of real life ant behaviour could be modelled and studied.

In a real-life ant colony, the pheromones laid by each ant can have different 'scents' indicitave of that ant's immediate action or environment. The trails used by the virtual ants here served only positive reinforcement of a path to the goal, but it would be interesting to see what the adverse effect an unsuccessful ant's pheromone trail would have on the performance of the optimisation. These 'negative' trails could alert following ants of nearby obstacles or dead ends. If implementing this, it would be prudent to note that a 'baseline' pheromone value should be associated with each node, and the pheromone evaporation would either decrease back towards this baseline following its use in a positive trail (ie if it is above the baseline pheromone value), or increase towards this baseline should its current value be below due to inclusion in a negative pheromone trail.

To get a more accurate and focused study of the optimisation capabilities of the Ant Colony Optimisation algorithm, the test environments remained the same across all iterations in any given attempt. At the end of each attempt, the iteration at which the best tour was discovered (ie the 'speed' of the optimisation) was recorded, as well as the number of ants to find the best tour on each succeeding iteration (showing the accuracy and consistency of the optimisation). Evaluating these two properties of the ACO algorithm allows further research into another interesting problem; if an obstacle should be introduced blocking the original path to the goal, how long would it take the ant colony to find the next best route? This would rely heavily on the speed and accuracy of the ants, but it would be interesting to see if finding the new route would be quicker as a result of preexisting pheromone trails hinting at the general direction of the goal. This research would feed heavily into the Emergency Response application of this algorithm which is discussed below.

An interesting investigation to pursue following on this project would be on the effect of multiple goal nodes on the behaviour of the ant colony. Having multiple goals of varying distance and direction from the home node would test the ants distribution, whether they balance themselves evenly between both goals or abandon the furthest goal in favour of the closer one which, inevitable, would produce a stronger pheromone trail per ant should the optimum route be found to both. The concept of weighted goals could also be introduced; inspired by the limited resources a piece of food in the real world would offer the colony. How long would it take for the ants to stop being influenced by trails to a goal no longer there? This would allow a lot more focus on the evaporation rate and its use in the `updatePheromones` stage in the ACO algorithm lifecycle.

8.2 Applications

The use of a non-exhaustive search within this project aims to expand the capabilities of Ant Colony Optimisation algorithms and apply it to new real-world cases.

Emergency Response

In the past, Ant Colony Optimisation algorithms have been utilised when designing traffic systems. This is suited to the original algorithms as one optimum solution was all that was required. The algorithm and its context in this project have opened up the opportunity to explore the colony's ability to find alternative routes and react to potential environmental changes - not needing to explore its entire environment. One possible application of this new capability would be for route calculation in emergency response vehicles.

Emergency Response teams rely on finding the fastest route to a point of crisis, in a time of crisis. The shortest path in terms of distance may not be the shortest route overall due to traffic or other road blockages. Using the algorithm and environment from this project, we could address this problem by inserting obstacles into the environment and having the virtual colony recalculate a new optimum, as described above in the Further Research section. The system in its highest level could function as follows

- A road network/town plan can be modelled in the existing simulation as a combination of obstacles representing any non-navigable surfaces with the remaining tiles representing roads or accessible routes.
- The system would receive inputs from sources identifying road closures, traffic congestion currently on specific routes, much like current navigation apps use when recalculating traffic routes.
- These inputs would translate into the environment by adding obstacles where complete impasses occur or a reduced pheromone value associated with road nodes carring heavy traffic.
- The ant colony optimisation algorithm could be run in this modelled environment, allowing for real-time introduction of new impasses or traffic statistics being accounted for when updating the pheromones or obstacles on any given iteration

This Use Case is dependant on the effectiveness of the algorithm when responding to changes in environment, but would be very useful to quickly find alternatives on the fly when in need.

Educational Resource

The visual nature of this project would lend itself well to an educational tool to showcase the performance of an AntColonyOptimisation algorithm. The use of the variables panel within the `Simulation`, the visual representation of the algorithm and it's route strengths over time allows users to gain a real-time understanding of what values affect the ACO algorithm, and how this algorithm operates along its entire lifespan. The test results gathered can be informative to future study on the Ant Colony Optimisation algorithm and its ability to be adapted into a new context, while including research on previous implementations of the algorithm gives users a full insight into the history and original intentions for the virtual ant colony.

The open plain matrix with a single goal approach would also be an effective environment to showcase the behaviour of algorithms such as A*, which thrive on similar shortest route problems.

9. Acknowledgements

This project draws inspiration from the Ant Colony Optimisation algorithm originally conceived by Marco Dorigo et al at IRIDIA in 1997. All code within this repository is our own, and we have made use of an AWS Elasticsearch cluster to store test results created from running this project. All result images are courtesy of the Kibana plugin within Elasticsearch.