The University of Nottingham

UNITED KINGDOM · CHINA · MALAYSIA

School of Computer Science

Joshua Argent

ID Number: 4245988

Supervisor: Dr Jason Atkin

Module Code: G53IDS

2017/18

The University of Nottingham

UNITED KINGDOM · CHINA · MALAYSIA

Multi-Agent Path Finding Techniques for Autonomous Air Traffic Control

Submitted April 2018, in partial fulfilment of the conditions for the award of the degree BSc (Hons) Computer Science.

Joshua Argent
4245988
School of Computer Science
University of Nottingham

Abstract

The aviation industry currently employs teams of highly skilled air traffic controllers to control the movement of air traffic and prevent mid-air collisions from happening. The men and women who perform this duty spend years training to do their job, but they are only human and can only handle a certain level of workload. The key aim of this project is to asses the feasibility of using a multi-agent path finding (MAPF) algorithm to automate the job of air traffic controllers.

A MAPF algorithm will calculate paths for agents (the aircraft) from a start location to a goal location on a graph, while maintaining a safe separation between them. There are many algorithms which can achieve this, each has its own pros and cons, however, the algorithm which this project uses is called Windowed Hierarchical Cooperative A* (WHCA*). This algorithm will find partial paths for agents within a windowed time period, which means it can adapt to a changing environment.

The project involves the development of two key components: a simulator to simulate the movement of aircraft, and an implementation of WHCA* to automate the control of aircraft agents. The project finally evaluates the effectiveness of different parameters and techniques, as well as the overall performance compared to human air traffic controllers.

Overall, the MAPF technique is effective at safely keeping aircraft separated, however, the limitation of the graph prevents more optimal routes for aircraft been found. With further research and development, the MAPF approach could be a feasible method for automating air traffic control.

Acknowledgements

I would like to take this opportunity to thank my project supervisor Dr Jason Atkin for his support and guidance throughout the project. Our weekly meetings really helped me to challenge myself and push the project further than I originally thought I could take it.

I would also like to thank my close friend, Mr Arian Moossavi, for his assistance with coordinate transformations necessary for development of the flight simulator.

Contents

1 Introduction

1.1 Project Overview

This project aims to explore the applications of multi-agent path finding (MAPF) algorithms for use in air traffic control. Multi-agent path finding algorithms, sometimes called cooperative path finding algorithms, are commonly used in games to navigate agents/entities through a network environment, while avoiding collisions with each other. There has been little research to see how effective they can be in a real world environment, or if they are suitable for automating the separation of air traffic, where an aircraft is considered an agent – this is what my project will investigate.

1.2 Brief History

In 1920 the first concept of air traffic control (ATC) was introduced at Croydon airport, London, using a radio and red or green signal light [1]. Since 1920, the number of air transport movements in the UK has increased from "below 200,000 in 1950" to "more than 2 million in 2001" [2]. Due to the increase in traffic and many high-profile mid-air collisions, radar technology was introduced to assist controllers with separating aircraft [3]. To this day, highly trained air traffic controllers use radar technology and radio to instruct the movement of air traffic. As explained later, there is little to no automation currently being used in this process.

The airspace in the UK is split into controlled and uncontrolled areas. Inside controlled airspace ATC will provide separation for traffic. There is a network of established 'airways' across the UK which act as corridors for aircraft to move along, "they are normally 10 miles wide and extend from a few thousand feet above ground to 24,500ft" [4]. Above 24,500ft, they are known as Upper Air Routes.

Based on data from services such as FlightRadar24 [5], it can be observed that aircraft don't always follow these airways, they sometimes cut corners and fly direct to a waypoint further along their route. Poor weather or congestion in some areas can also cause aircraft to need to deviate from the airways.

1.3 Aims and Objectives

The aim of the project is to implement a multi-agent path finding technique to automate the control of en-route air traffic in an airway system. The project will achieve this aim by meeting the following objectives:

A. Develop a flight simulator that simulates the movement of multiple aircraft on a GUI radar display. It should display summary data, such as flow rate and efficiency, which can be used to compare different separation techniques and parameters.

B. Model the airways system as a graph data structure that can be used by path finding algorithms.

C. Implement variants of a multi-agent path finding algorithm to automate the control of aircraft while maintaining safe separation.

D. Detect and overcome deadlock situations.

E. Evaluate what parameters and techniques achieve the most efficient use of airspace and fuel/time.

1.4 Motivation

This project investigates the use of MAPF techniques to reduce the workload for en-route air traffic controllers or potentially remove the need for controllers at all. Typically a controller will be responsible for 10 to 15 aircraft [6]. Automation has the potential to allow controllers to oversee more aircraft at once and increase airspace capacity; NATS (formerly National Air Traffic Services) implemented a tool to predict aircraft positions ahead of time which lead to an average 15% increase in capacity [7]. This has obvious benefits for the airspace authorities as they will not need to train and employ as many staff.

The potential for more efficient use of airspace should not be overlooked. Using the latest advancements in MAPF, more aircraft may be able to use the same amount of airspace yet still maintain safe separation. There is a growing problem, particularly in the UK, where airspace is becoming over congested which leads to flight delays and cancellations. NATS explains that lack of integration between different parts of the air traffic system leads to congestion [8]. I believe that an automated system could help to eliminate this problem.

The success of the project will be determined by whether a conclusion can be drawn about the feasibility of using MAPF as an aircraft separation technique. A positive conclusion can be made if the final system can handle at least as many aircraft as human air traffic controllers and achieve equal or more efficient routes.

1.5 Dissertation Structure

Related Work

An overview of the artificial intelligence currently used in air traffic control and related research. Also analyses a range of existing MAPF algorithms and their properties.

Methodology

A description of the strategy for development of the project, including the choice of algorithms and overall structure for the system. Presents a general outline for how the system should be tested.

Design

An explanation of the system design and the reasoning behind the design decisions. Includes a design for the overall structure, system specification, class diagrams and user interfaces.

Path Finding

Explains in detail how the multi-agent path finding algorithm will be designed and implemented. Demonstrates how airways can be modelled as a 3D graph data structure that pathfinding algorithms can use. Also proposes a novel method for detecting deadlocks.

Implementation

A description of the implementation for the key elements of the system. Identifies items which had to be implemented differently to the original design or areas which were missing.

Evaluation

Compares the effectiveness of different parameters for handling a high volume of air traffic. Also compares the performance of the system to human air traffic controllers.

Summary and Reflections

Summarises the success of the project and how well the project management went. Proposes areas which could be further researched and developed that may benefit the industry.

2 Related Work

This section explores two distinct areas: current and proposed automation tools for air traffic controllers and multi-agent path finding algorithms (also known as cooperative path finding algorithms).

2.1 Automation in Air Traffic Control

NATS, who provide air traffic control in the UK, already uses a system called I-FACTS (interim Future Area Control Tools) which will calculate the trajectories of aircraft up to 18 minutes in advance, and, if a conflict is likely, warn the controller [7]. This better enables controllers to be able to visualise aircraft positions in the future.

ATC aims to maintain separation by issuing instructions to aircraft in order to de-conflict them. These instructions could be a change in heading, change in speed or change in altitude [9]. It could also involve sending an aircraft along a different route, however, this would require longer-term planning. It is worth noting that assigning a heading to an aircraft creates a level of uncertainty because the actual track/path of the aircraft is influenced by the wind, which is variable. Instructing an aircraft to fly to a waypoint is more favourable because it reduces uncertainty [10].

One example of a conflict resolution algorithm has been developed by Durand et al [11] which uses a genetic algorithm to de-conflict en-route aircraft. It works by grouping and deconflicting aircraft which are near to each other. For example, “if aircraft A conflicts with aircraft B and aircraft B conflicts with aircraft C, then aircraft A, B, and C are aggregated into the same cluster. Each cluster is then deconflicted independently, using a genetic algorithm.” [9] This divide and conquer style approach proves to be a good way of reducing the runtime of the algorithm.

One of the biggest problems Durand et al faced was balancing the uncertainty in the prediction. The uncertainty level increases with time, which makes issuing conflict resolution instructions in acceptable time difficult [11]. Therefore, every effort should be made to either reduce uncertainty or select an algorithm which can handle uncertainty,

§2.2 explains that 'online' algorithms can handle some level of uncertainty.

Finally, Lehouillier et al [9] proposes some performance indicators which may be useful for comparing different separation techniques:

- "The entering flow per hour for a sector"
- "The number of conflicts"
- "The number of conflict-resolution manoeuvres" [9]

Obviously, the ideal algorithm will be able to have a high flow rate, zero conflicts and as few conflict-resolution manoeuvres as possible. These metrics will be a good tool to use for gauging how successful my implementation is.

2.2 Cooperative Path Finding Algorithms

Silver describes the cooperative path finding problem as "a multi-agent path planning problem where agents must find non-colliding routes to separate destinations, given full information about the routes of other agents" [12].

Cooperative path finding algorithms are generally put into two classes depending on their ability to adapt to a changing environment. An off-line algorithm calculates full paths for all agents at the beginning and does not monitor progress, where as an online algorithm will periodically recalculate paths so it can adapt to change easily.

The 'off-line' methods will take the list of agents, their start and goal nodes, and then calculate a set of complete paths for each agent. Once the route is calculated, the agents follow their path and no further calculations are made [13]. This method is not flexible to change, if the environment or agents change, any calculated paths will become invalid.

The 'online' methods will periodically calculate the agent's paths to their goals many steps in advance, but not the whole route. The advantage to this method is that if anything changes, the partial paths can be recalculated again. This makes online algorithms tollerant to uncertainty in agents or environments. It also reduces the computation time because there is a limit in place, this may mean that the result is not complete or optimal, however [13].

Algorithm Review

I will now review some existing cooperative path finding algorithms that are applicable to this project.

Local Repair A* The first algorithm I will review is Local Repair A* (LRA*), which is commonly used in computer games [12]. LRA* is an online path finding method [13] where each agent will use the A* algorithm to find the shortest path to its goal and then start following that path, regardless of other agents. If a collision is about to happen, one of the agents will recalculate its path around the offending agent. The drawback of this technique is that deadlocks and cycles are very common. Some variations use an 'agitation level' to prioritise agents that have had to take a lot of avoiding actions [12].

Cooperative A* An alternative off-line algorithm is Cooperative A* (CA*), where each agent searches for a path, using A*, through a 3D space-time data structure called a reservation table. Once an agent has planned its route, it is marked in the reservation table to prevent other agents from colliding with it. The reservation table is a very sparse data structure so should be implemented as a hash table using the key (x, y, t). The order in which agents perform their searches influences who gets the priority [12].

Hierarchical Cooperative A* Hierarchical Cooperative A* (HCA*) takes CA* a step further and "uses a simple hierarchy containing a single domain abstraction, which ignores both the time dimension and the reservation table" [12]. It then uses these abstract distances as a heuristic to guide the search, essentially ensuring that the paths found are as optimal as possible [12].

Windowed Hierarchical Cooperative A* Silver developed the Windowed Hierarchical Cooperative A* (WHCA*) algorithm, which is an online variant of HCA*. WHCA* solves the issues of unfair agent priorities and the need for a complete route to be calculated before any agent can move. The basic principle is that instead of calculating a whole route through space-time, a route is only calculated to a fixed depth or 'window'. The agents begin to follow their partial route and periodically the manager will calculate further partial routes. To prevent agents going down dead ends, the algorithm will calculate a full route through space (using A*) and then perform the cooperative search in the space-time dimension [12].

The heuristic that Silver uses in WHCA* is called 'true distance' which is the length of the shortest path to the destination. In other words, the length of the shortest spatial A* path. This is a good heuristic because the agent will always search down the shortest path, unless another agent will block it [14].

Heuristic Review

An informed search algorithm, such as A*, can be implemented in two key ways: as a tree search or as a graph search. A tree search is able to revisit nodes during a search if a better path can be found to it. A graph search has a 'closed list' which visited nodes are added to, closed nodes can not be visited again. A graph search requires more memory (for the closed list data structure) but can have a faster runtime than a tree search [15].

A heuristic is said to be admissible if it holds the property $h(n) \leq h^*(n)$, where $h(n)$ is the estimated cost from the current to the goal node, and $h^*(n)$ is the optimal path cost. Essentially, an admissible heuristic never overestimates the cost to the goal, such as straight line distance. If an admissible heuristic is used in a tree search, it will always find one of the optimal paths [15].

To find an optimal path with a graph search, however, the heuristic needs to be consistent as well as admissible. A consistent heuristic holds the property $h(A) \leq cost(A, B) + h(B)$ between nodes A and B [15]. True distance is an example of a consistent heuristic [14].

Selecting a heuristic which can reliably deliver an optimal route is important because it means aircraft will take less time to reach their destination which will help to increase airspace capacity.

A final consideration when developing the aircraft separation algorithm is the 'Nash equilibrium'. Turocy et al explains that "Nash equilibrium recommends a strategy to each player that the player cannot improve upon unilaterally, that is, given that the other players follow the recommendation. Since the other players are also rational, it is reasonable for each player to expect his opponents to follow the recommendation as well" [16]. This theory can be applied to a situation of two aircraft; where it may be more beneficial overall for one plane to inconvenience itself and perhaps slow down to allow another to pass in front. The plane slowing down has no incentive to do so but it will benefit the other traffic.

3 Methodology

3.1 Development Process

I decided to split the project into two separate parts:

A. A simulator which will simulate the movement and control of aircraft, provide a graphical radar display of aircraft and also expose a programmable interface to control aircraft.

B. An 'AI module' which will interface with the simulator and control the aircraft, based on the calculations of a cooperative path finding algorithm.

I chose to create a bespoke simulator instead of using a third party tool, as there is not currently any existing product that does exactly what I require. Existing software would need either serious modification or be too complex for my requirements. One open source simulator, called OpenScope [17], looked like a good starting point, however, it heavily focussed on the airport approach control as opposed to the overall en-route control. It would have required a lot of modification to make it meet my requirements.

3.2 Choosing a Path Finding Algorithm

An off-line algorithm can not handle an unexpected change in the environment or agents because it does not monitor agent progress [13]. The nature of aircraft and flight can be unpredictable, even the most realistic flight model can not make perfect predictions, which makes off-line searches unsuitable. Therefore, an online algorithm, such as LRA*, would be better.

The commonly used Local Repair A* algorithm would be the easiest method to implement, however there is a real danger of deadlocks and cycles happening [12] which makes it unsuitable. For this reason I chose to use an online variation of Cooperative A* (such as WHCA*) because there is a much lower chance of deadlocks and dead-ends [12]. There will still be a need for an extra deadlock detection mechanism in order to recover from these infrequent deadlocks, which will probably involve delaying aircraft. Therefore, it is important that the algorithm I use is not too susceptible to them, as otherwise the overall efficiency would be effected.

3.3 Testing

To gauge how successful a MAPF strategy is for automating air traffic control I need to compare the performance of the AI to the performance of human air traffic controllers. Real world data for aircraft can be acquired from many different sources. The aircraft type, start location and end location can be identified and then fed into the MAPF system which will try to safely navigate the aircraft to their goal, under the same conditions the human controllers faced. The optimality can then be compared to the original data to see how effective MAPF is.

4 Design

4.1 Determining the Level of Abstraction

Simulator

It is important that the simulator is able accurately simulate the movement and behaviour of aircraft as they would in the real world. Therefore, the aircraft should be able to act independently to the rest of the system, so even if the AI were to fail, the aircraft would continue to move. Much like in real life, if an aircraft loses communication with ATC they will continue to fly.

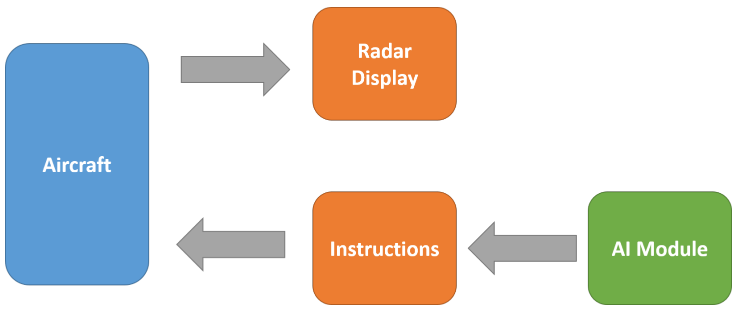


Figure 1: Simulator abstraction

Figure 1 shows how the simulator component integrates into the rest of the project. The AI module communicates to the aircraft through the simulator's API by sending instructions to aircraft, such as climb, descend or turn. The aircraft will follow these instructions, the result being displayed on a 'radar like' display.

Cooperative Pathfinding

I decided to also abstract the path finding algorithms away from the aircraft navigation. The advantage of doing this is that it can be independently tested in a much more controlled environment with less uncertainty in agent behaviour. Figure 2 shows a class diagram for the design of the path finding module. The specific implementation details are discussed later, however, the unit is split into two main parts; the path finding algorithms and the agent manager. Each component is designed to be as generic as possible to allow the final implementation to override and extend the functionality, as appropriate.

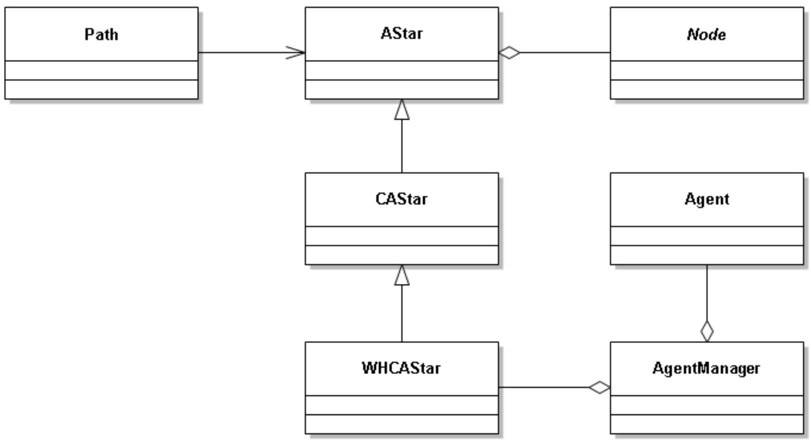


Figure 2: Multi-agent path finding class diagram

4.2 Specification

Simulator Specification

Through my research so far, I have determined that the simulator has the following requirements:

1. Have a radar display that shows the positions of aircraft in relation to waypoints and airways.
2. Be able to load airway and waypoint data from a plain text file.
3. Each aircraft should have an information label that shows the aircraft's callsign, aircraft type, ground speed and altitude.
4. There should be a menu where the wind speed and direction can be set.
5. All aircraft, along with their details, should be displayed in a list.
6. The whole simulator should be abstracted into an API that the separation algorithm will be able to communicate through:
   (a) Should be able to add/remove aircraft.
   (b) Should be able to issue instructions including heading assignment, direct-to-waypoint, altitude, speed, hold to aircraft.
   (c) Be able to access aircraft details and position.
7. The simulator should have a slider to speed-up/slow-down or pause/resume it.
8. Should be able to detect conflicts between aircraft. Conflicting aircraft should be highlighted in red.
9. Should display flight statistics including:
   (a) Number of aircraft in the scenario.
   (b) Flow rate.
   (c) Number of conflicts.
   (d) Number of conflict resolution manoeuvres (CRMs) – i.e. the number of instructions sent.

AI Module Specification

The AI module should be able to:

1. Load a list of 'flight plans' from file which specify an aircraft, the entry and exit point in the scenario and the time it will enter.
2. There should be a form to manually add aircraft to the scenario by entering the same data as above.
3. It should implement an effective cooperative path finding strategy to direct aircraft from their entry point to their exit point while maintaining safe separation.

4.3 Design of the Simulator API

Continuing with the idea of abstracting the simulation of aircraft from the controlling AI, the simulator needs an application programming interface (API) for the AI module to interface with. This section specifies the design for this using class diagrams. The full class diagram that this section refers to can be found in Appendix A.

Instructions

The instructions that can be given to aircraft can be generalised into a generic *Instruction* abstract class that can be implemented by each instruction type. The instructions also make use of an 'observer pattern' so that interested parties can be notified when an instruction is complete. I created five concrete implementations of *Instruction*:

- Heading instruction – instruct an aircraft to turn onto a particular heading (0 to 360°).
- Altitude instruction – climb or descend to a given altitude.
- Speed instruction – instruct an aircraft to accelerate or decelerate to a given speed.
- Waypoint instruction – instruct an aircraft to fly directly to a given waypoint.
- Hold instruction – make an aircraft circle in its current position.

Waypoints, Airways and Scenarios

The AI module and radar display will need a data structure to contain the data for navigational waypoints and airways. Everything is encapsulated within a *Scenario* object which contains a map of waypoint names to *Waypoint* objects and a list of *Airways*. An airway is defined as being between two waypoints with an upper and lower altitude.

Simulator Singleton

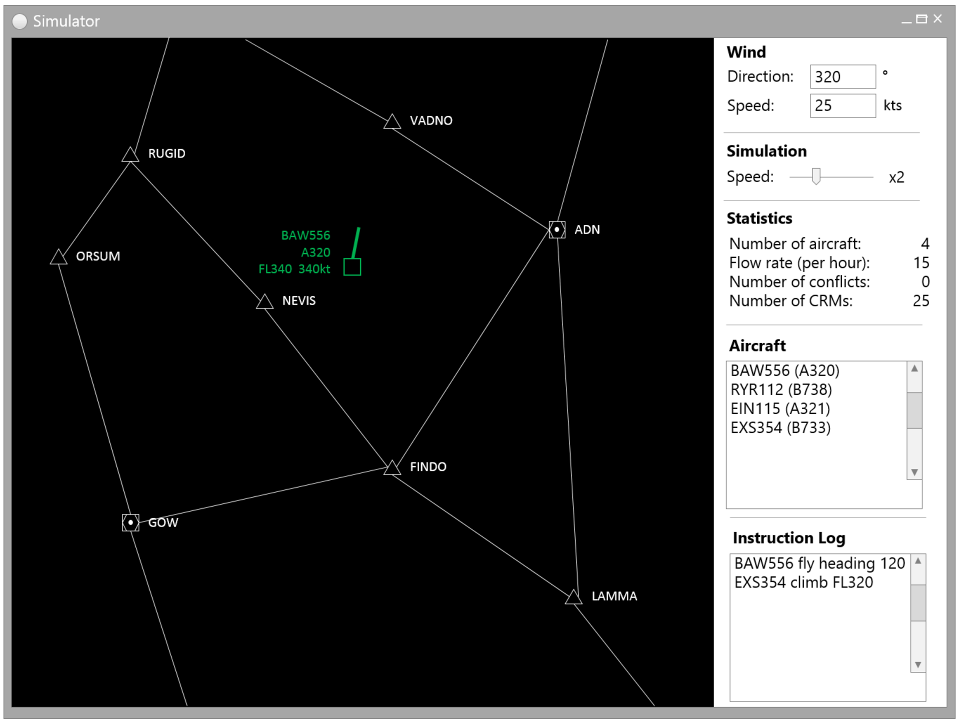
There is a Simulator singleton class, which will contain the *Scenario* and all the *Aircraft*. It is responsible for managing the movement of aircraft, updating the graphical user interfaces and is the main entry point for the AI module to communicate with the simulator.

Aircraft

The final component to the simulator API is of course the *Aircraft* class, which represents a single aircraft moving in the simulator. It contains basic properties such as heading, speed and altitude and can execute instructions which are passed to it. All the properties are protected to ensure that nothing can alter the values outside the simulator – only instructions need to change these properties. Each *Aircraft* has an *AircraftProfile* object reference which specifies basic properties, such as maximum speed and climb rate, for the particular type of aircraft. A number of static *AircraftProfile*s for common aircraft types have been defined.

4.4 User Interface Design

The purpose of the user interface is to allow the user to control the basic settings of the simulator (simulation speed and wind properties) and to also visually observe aircraft behaviour.



Wind
Direction: 320 °
Speed: 25 kts

Simulation
Speed: x2

Statistics

Number of aircraft:	4
Flow rate (per hour):	15
Number of conflicts:	0
Number of CRMs:	25

Aircraft
BAW556 (A320)
RYR112 (B738)
EIN115 (A321)
EXS354 (B733)

Instruction Log
```
BAW556 fly heading 120
EXS354 climb FL320
```

Figure 3: User interface design for the simulator

Figure 3 shows the design of the graphical user interface for the simulator. The majority of the screen is filled with a radar display that shows the network of airways and waypoints and the position of the aircraft. The radar display is zoom-able and pan-able using the mouse and scroll-wheel, this is necessary because there will likely be too much information to display at once. On the right hand side of the window is a menu with options for controlling the simulator scenario. The actual addition, removal and control of aircraft will be done by the AI module.

5 Path Finding

The path finding algorithm and associated functionality is what makes the AI module. This section describes the implementation details of the algorithm and any specific changes which needed to be made for it to be able to separate aircraft.

5.1 Windowed Hierarchical Cooperative A*

I decided to implement WHCA* as generically as possible to allow me to thoroughly test the basic algorithms before adding the extra complication of aircraft. Each algorithm extends the implementation of the previous. *CAStar* introduces the reservation table and a function to reserve nodes at a given time. It also overrides the *getSuccessors()* function that *AStar* uses, instead of returning all the successor nodes, it will only return successors which are not blocked in the reservation table at the given time. *WHCAStar* overrides *CAStar*'s reservation function to only make reservations that are within the specified window. Figure 4 shows the class hierarchy for this abstraction. All of the algorithms implement the *IGraph* interface, which just specifies a generic interface for a graph.

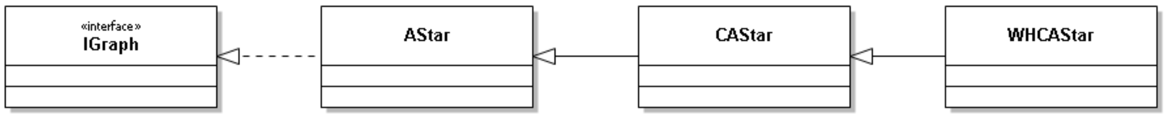


Figure 4: WHCA* abstraction class diagram

Data Structures

My implementation of A* uses a Java *HashMap* to map a *Node* to a list of successor *Node*s. Java *HashMap*s are an implementation of hash tables which have a fast average lookup time complexity of $O(1)$ and a space complexity of $O(n)$, where n is the number of nodes [18]. This makes them an ideal data structure for the A* algorithm, which involves a lot of lookup operations. The graph is also expected to be very large, therefore a hash table is a more preferable data structure compared to an adjacency matrix, which has a space complexity of $\Theta(n^2)$ [18].

5.2 Deadlock Detection

The IEEE defines a deadlock as "a situation in which two or more tasks are suspended indefinitely because each task is waiting for a resource acquired by another task" [19]. This is in the context of parallel programming, however, the same sort of situation can occur in a MAPF algorithm. A deadlock could arise if an agent, agent A, is unable to make any progress because it's next node is blocked by agent B, which can also not move because it's next node is blocked by agent A. This type of scenario can be very difficult to detect and can be equally difficult to overcome.

From my observations, most deadlock scenarios arise when the search algorithm can not find a path to the goal, essentially all the possible routes are blocked by other agents. This can be seen in Cooperative A* when too many nodes have been reserved by other agents, resulting in no possible route to the goal. A symptom of this kind of deadlock is when both the open and closed lists in the underlying A* algorithm are the same between iterations. If this scenario occurs, it is safe to assume that the search has reached a dead end that it cannot recover from.

My implementation of A* performs a deadlock check every time a node is popped from the open list (Figure 5). It compares the hash codes of the open and closed lists from the current and previous iterations, if they match then the search is in a deadlock and can be terminated.

```
protected boolean isInDeadlock()
{
   int hashCode = (closedList.hashCode() / 2) + openList.hashCode();
   if(previousHashCode == hashCode)
   {
      return true;
   }
   previousHashCode = hashCode;
   return false;
}
```

Figure 5: Detecting a deadlock between iterations of A*

After a deadlock occurs, the search will throw an exception that is caught by the agent manager. An explanation of how the deadlocks are handled can be found in §5.3.

5.3 Agent Manager

My implementation of WHCA* will not manage agents on its own. I implemented a class called *AgentManager* that acts as a central manager for handling agent scheduling, calculating paths using WHCA* and distributing the paths to the agents. Along with the *AgentManager*, there is an *Agent* class which specifies abstract methods that an agent should implement, such as its current position, its goal node and its priority. The manager behaves in a very generic way, it bases all the scheduling on an agent's priority value and distributes the calculated paths through the *Agent*'s abstract *runPath()* method. The advantage of abstracting the agents and agent scheduling away from the intricacies of aircraft specific properties is that the functionality can be tested independently as separate units.

Another advantage to this abstraction is that it offers a degree of modularity, certain parts can be overridden and changed easily without effecting the original implementation. One such example is the priority system for scheduling agents; there are different systems that can be tested, such as a first-come first-served strategy (FCFS) or prioritising agents furthest from their goal. These options are tested and discussed later in the evaluation.

The last role of the agent manager is to handle deadlock exceptions thrown by WHCA*. When a deadlock occurs, the agent manager will first reserve the agent's current position in the reservation table, thus protecting it from conflicting with other agents. It then gives the agent a path that simply instructs it to stay in it's current position, in the context of aircraft it issues a hold instruction. Hopefully, in the next WHCA* window, the reservations, agents and priorities will have changed enough to allow it to successfully find a path.

5.4 Modelling Airways as a Graph

The airways need to be represented in a graph data structure that can be used by WHCA*. The way this is done will have a huge impact on the effectiveness and optimality of the routes that aircraft are given. For example, aircraft fly in three dimensions so the graph should consider this, otherwise the majority of the airspace will be wasted. Another thing to consider is that some airways cross each other but don't have a waypoint at the intersection, this needs to be included in the model as a node. Finally, it is a good idea to split the long graph edges into smaller sub-sections to allow multiple agents to use the same airway at once. Figure 6 shows a particularly complex section of airways in Scotland.

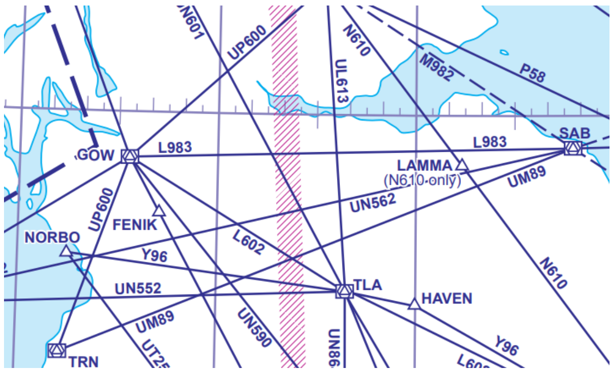


Figure 6: A section of airways in Scotland

Identifying Airway Intersections

The first part of my algorithm takes the airway data and maps it into a graph object using the airways as edges and the waypoints as nodes. This object will be used as the basis for further optimisations. The next step identifies intersections where nodes do not exist. Algorithm 1, takes an open list of graph edges, pops an edge and iterates over all the other edges to identify ones which intersect. If they do, it creates four new edges and a node to join them together, then adds them to the open list. Once the open list is empty, the algorithm is complete – all the intersections have been replaced by nodes.

Algorithm 1 Find-Intersections(*open*)

```
Require: $open$ is a stack of graph edges
 1: $edges \leftarrow \theta$
 2: while $opent \neq \theta$ do
 3:     $P \leftarrow open.pop()$
 4:     $hasIntersections \leftarrow$ false
 5:     for all $Q \in open$ do
 6:         if (not Are-Connected($P$, $Q$)) and Intersect($P$, $Q$) then
 7:             $W \leftarrow$ Intersection($P$, $Q$)
 8:             $open \leftarrow$ Edge($P.from$, $W$)
 9:             $open \leftarrow$ Edge($W$, $P.to$)
10:             $open \leftarrow$ Edge($Q.from$, $W$)
11:             $open \leftarrow$ Edge($W$, $Q.to$)
12:             $open.remove(Q)$
13:             $hasIntersections \leftarrow$ true
14:             break
15:         end if
16:     end for
17:     if not $hasIntersections$ then
18:         $edges \leftarrow P$
19:     end if
20: end while
21: return edges
```

Adding Intermediate Nodes

The next part identifies long edges which could be split into smaller edges. I implemented an algorithm, Algorithm 2, that uses divide and conquer to recursively split an edge into smaller edges until they are a more suitable length.

Algorithm 2 Intermediate-Points(A, B, numberSegments)

```
Require: $2^n = numberSegments$
1: if $numberSegments = 2$ then
2:     return Midpoint(A, B)
3: else if $numberSegments > 2$ then
4:     $mid \leftarrow$ Midpoint(A, B)
5:     $left \leftarrow$ Intermediate-Points(A, mid, numberSegments / 2)
6:     $right \leftarrow$ Intermediate-Points(mid, B, numberSegments / 2)
7:     return $\{left, mid, right\}$
8: end if
```

Altitude

Finally, the graph needs to be given a third dimension by considering the altitude. Aircraft need 1000ft of vertical separation, so a copy of the airway graph is made at each 1000ft interval up to the vertical limit of the airway. Each level should then be given branches going up and down to the levels above and below (as shown in Figure 7). The branches aren't limited to just connecting adjacent levels, they can branch to several levels deep. The advantage of this is that it will allow aircraft to climb/descend multiple levels at once, giving them more options to avoid conflicts.

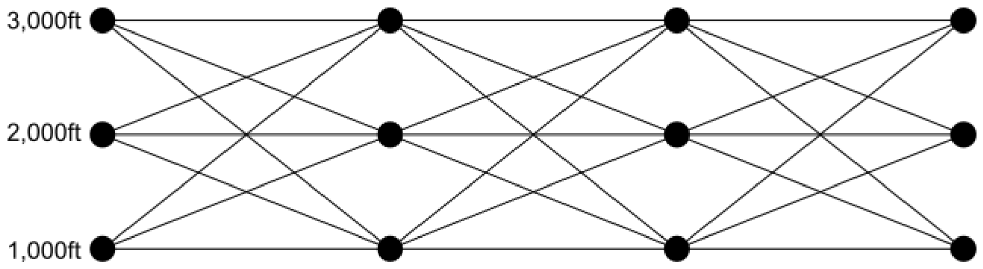


Figure 7: Cross-section of the 3-dimensional graph

5.5 Spatial Nodes

The system builds the graph from *SpatialNode*s which are an implementation of the abstract *Node* class, used by *AStar*. A spatial node is a node within 3D space and is defined using a latitude, longitude coordinate and an altitude. *SpatialNode* is extended further by three concrete implementations:

- *SpatialNodeWaypoint* – a node at the same location as a navigational waypoint
- *SpatialNodeIntersection* – a node that is used at the discovered airway intersections
- *SpatialNodeIntermediate* – a node that is used as intermediate points on a particularly long airway

SpatialNode implements heuristic and cost functions based on an estimate of the flight time for the aircraft. Initially, the heuristic used is the straight line time estimate from the node to the goal node. In the evaluation section I present alternative heuristics and compare their effectiveness.

Using Time as a Heuristic

The reason for using time as a heuristic, instead of distance, is because there are several factors that will effect flight time over a constant distance. The most influential is altitude; the higher an aircraft flies, the faster it travels [20]. This will encourage aircraft to climb to higher altitudes, as they will be able to achieve shorter flight times. The aim is to find the quickest route, which is not necessarily the same as the shortest distance route.

It is implemented using a function called *estimateFlightTime()*, which will estimate how long it will take for an aircraft to fly between two points. This function is also responsible for determining whether the aircraft can climb/descend to the given altitude or if the performance of the aircraft does not allow it. If it is not possible, it will return a very large number that will prevent the node being explored further by WHCA*.

5.6 Testing

In order to test the multi-agent path finding algorithm, I needed a simplified environment to work with that doesn't have the complication of aircraft or three dimensions. I decided to create a 10 by 10 grid GUI (Figure 8) that lets the user select which squares are nodes (white) and which are barriers (black). Multiple agents start and goal locations can be specified, along with the WHCA* window size, the algorithm can then be ran a single step at a time in order to systematically see how it is working.

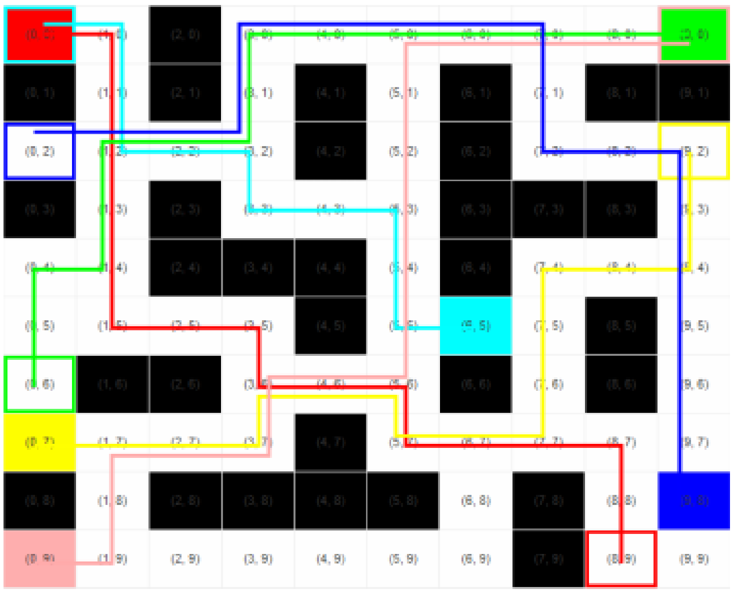


Figure 8: Grid GUI with 6 agents, the filled squares are the agents and the corresponding coloured border squares are their goals. The paths that the agents took have been marked on for clarity.

Along with this integration testing, I wrote some unit tests in JUnit to test the A* implementation and the graph building functionality. These tests were helpful for ensuring that faults were not introduced during development.

6 Implementation

6.1 Overview

The implementation of the system is quite complex as there are several key components which make up the core functionality. Figure 9 shows a diagram of the system, showing the boundaries between simulator and AI modules and the communications between components.

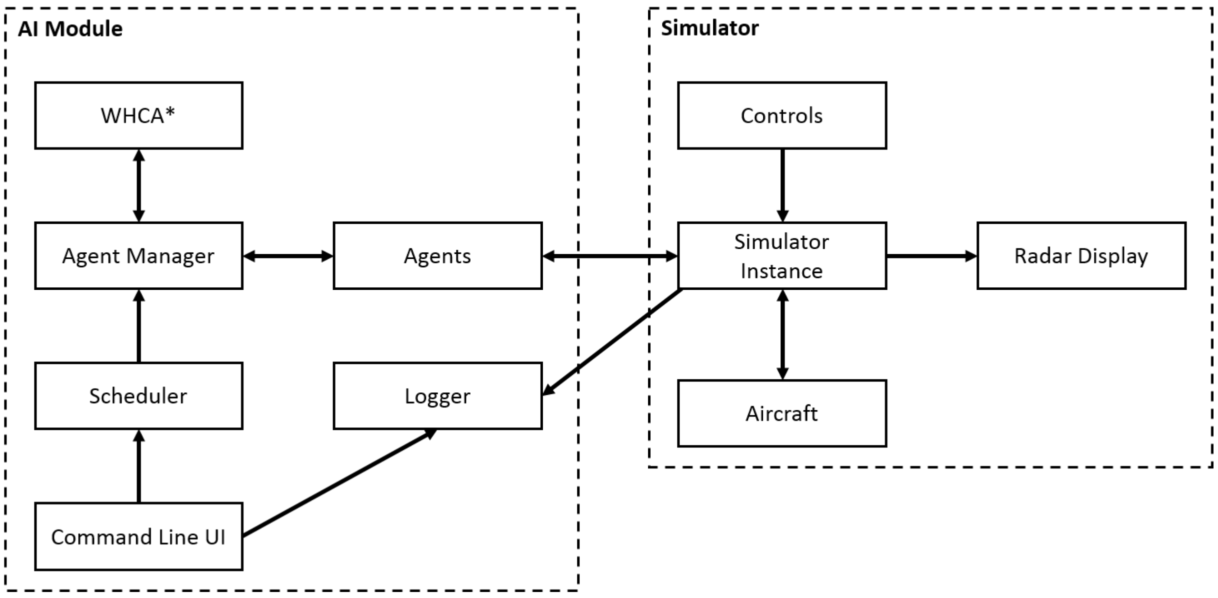


Figure 9

Command Line UI This lets the user control the AI module through a command line interface. It is primarily used to control the scheduler and logger.

Scheduler Loads a schedule file and spawns a thread that will create new agents at the times specified in the schedule. See §6.3.

Agent Manager Iterates through all the agents, in the order of their priority, and uses WHCA* to calculate cooperative paths. It then distributes these paths to the agents.

WHCA* Implementation of WHCA* path finding, including the graph building functionality.

Agents Each agent is associated to one aircraft. It is responsible for passing ATC instructions to the simulator based on the calculated paths. When a path is complete, it will request that the agent manager calculates the next window of paths.

Logger Spawns a thread to periodically log the positions of all the aircraft in the simulator. See §6.4.

Simulator Instance A singleton which encapsulates the core functionality of the simulator. Runs a thread to periodically update aircraft positions.

Aircraft Represents the state of a single aircraft.

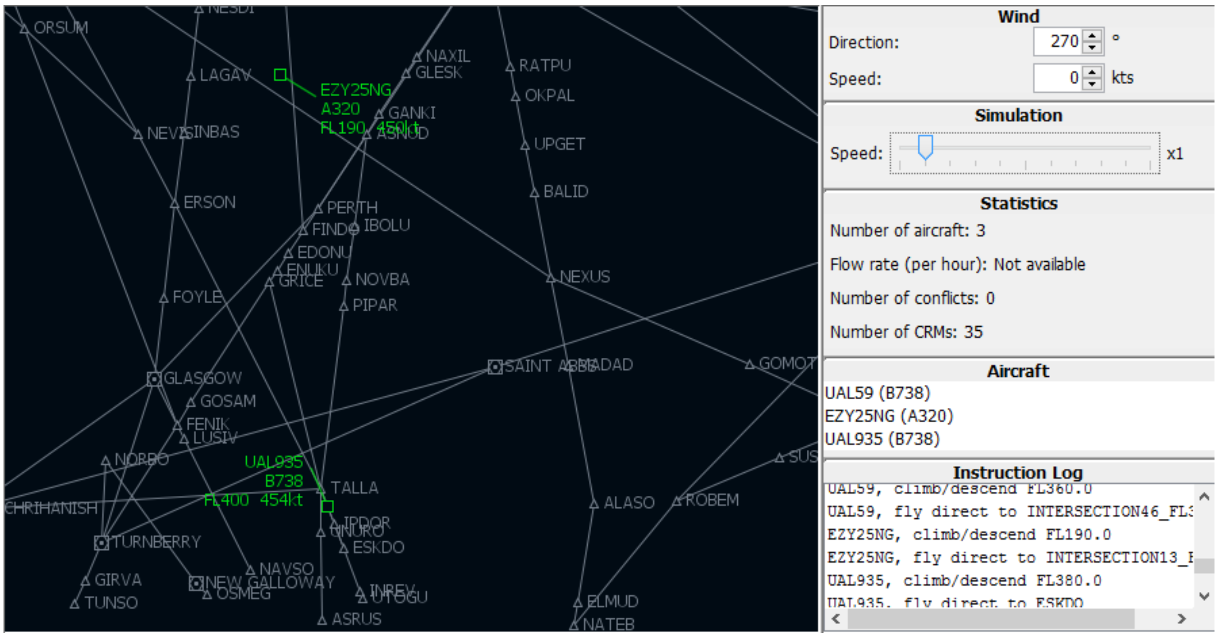
Controls A GUI for controlling the simulator (e.g. simulation speed).

Radar Display Displays all the aircraft, airways and waypoints on a GUI radar style display.

6.2 Simulator

The flight simulator has been implemented in Java, as described in the design section. I chose Java because it is well suited for object-oriented programming, which fits the nature of the project – specifically the separation of the simulator and AI into independent modules. The simulator has two key threads – the Swing UI thread for displaying the radar, shown in Figure 10, and a separate thread to handle the movement of the aircraft. This ensures that, even if the aircraft thread becomes blocked, the user interface thread will still be interactive and vice versa.

The aircraft positions are in the context of latitude, longitude and altitude, therefore, all movement needs to be calculated using polar coordinates. Veness has published a collection of formulas for calculating various things using latitude and longitude coordinates, such as the mid-point between two points, or the distance between two points [21]. These formulas proved to be extremely helpful for calculating aircraft movement; each aircraft has a direction and speed which is used to calculate the next position in t amount of time. This constant update of positions happens in an infinite loop, inside the aircraft thread.



Wind

Direction: 270 °

Speed: 0 kts

Simulation

Speed: x1

Statistics

Number of aircraft: 3

Flow rate (per hour): Not available

Number of conflicts: 0

Number of CRMs: 35

Aircraft

UAL59 (B738)
EZY25NG (A320)
UAL935 (B738)

Instruction Log

```
UAL59, climb/descend FL360.0
UAL59, fly direct to INTERSECTION46_FL3
EZY25NG, climb/descend FL190.0
EZY25NG, fly direct to INTERSECTION13_I
UAL935, climb/descend FL380.0
UAL935, fly direct to ESKDO
```

Figure 10: Simulator

6.3 Flight Schedules

The system has the ability to load a ‘schedule’ of flights; which defines a flight’s start and end location, aircraft type, start and end altitude and the time at which it should be scheduled for. The system will then follow the schedule and automatically add aircraft based on the time they are scheduled for. The reason I implemented flight schedules was so that I could run the same scenario of aircraft on different variations of the system, which allows for comparisons to be made. The schedules are defined in a JSON text format, because it is easy to create and edit, Figure 11 shows an example schedule entry.

```
[
 {
   "callsign": "EZY538A",
   "type": "A320",
   "entry_time": 12,
   "entry": "NATEB",
   "exit": "RATSU",
   "entry_altitude": 10000,
   "exit_altitude": 39000
 }
]
```

Figure 11: Example JSON flight schedule data

6.4 Flight Data Logger

It became apparent that there was a need to be able to record the positions of aircraft for analysis, which the original specification did not include. I decided early in the development phase that I needed a way to compare different parameters and algorithms in a controlled way, I was particularly interested in comparing the optimality of the routes that aircraft take. I implemented a logger thread which independently communicates with the simulator to generate a GPS track file (.gpx) for each aircraft. GPX files are an XML format that records a latitude, longitude and altitude at regular time intervals. I chose this format because there are many programs available, such as MatLab, which can open GPX files and analyse the data.

Along with the flight GPS data there was a need to record the changes of priority for the aircraft. I added a function that listens to the agent manager and will write the agent priorities to a log file, every time the priorities change. This allowed me to compare how the agent priorities change over time.

One final addition was the option to create a log when two aircraft conflict. The log file contains the details of the aircraft; including speed, altitude, heading, agent priority, current and goal nodes. It also dumps the contents of the WHCA* reservation table. During the development this provided a useful way to analyse what was causing conflicts, such as the block altitude problem discussed later in this chapter.

6.5 True Distance Heuristic

Initially, I used a straight line distance heuristic in my path finding algorithm. The straight line distance simply measures the distance from the node to the goal node, which works well as a heuristic because A* will favour nodes nearest to the goal. However, WHCA* should use a 'true distance' heuristic [14] to guide the search. In other words, the heuristic cost of each node should be the actual cost of a path to the goal, ignoring any other agents. This means that the algorithm will always find the shortest route and only deviate from it to avoid other agents.

My initial approach was to use Dijkstra's algorithm which will find the length of the shortest path between two given nodes in a weighted graph [18]. However, the computational cost of running this algorithm every time the heuristic needed to be calculated was unfeasible.

One solution was to cache any calculations, but I found that a more complete and robust solution was to use the Floyd-Warshall algorithm, as described in Cormen's Introduction to Algorithms [18], to calculate the distance of every shortest path between every node during the program's initialisation. The Floyd-Warshall algorithm builds an n by n sized matrix of shortest distances between nodes, where n is the number of nodes. It has a runtime of $O(n^3)$ so it takes a few seconds to build the matrix, however, once it is complete, all that is required in order to find the shortest path distance is a lookup in the matrix - which has runtime $O(1)$ [18].

My implementation doesn't use the full three dimensional graph because the huge number of nodes would result in an excessively large matrix. Instead, I give the Floyd-Warshall algorithm a version of the graph that has been flattened into 2D space – without any altitudes. It also stores distance values in a 16-bit short, which helps to reduce the amount of memory used. When a heuristic calculation is made, the following steps are carried out:

1. The pre-calculated shortest distance between the two nodes is fetched from the Floyd-Warshall matrix.
2. The change in altitude, if any, is calculated.
3. The flight time for the distance is estimated, accounting for the altitude, and returned as the heuristic cost.

6.6 Block Altitude Reservations

One common cause for conflicts was when aircraft were descending and climbing. The problem was that agents would reserve the nodes they are climbing or descending to but not reserve the nodes above/below, leaving them exposed to other aircraft coming closer than 1000ft vertically. A conflicting scenario caused by this issue can be seen in Figure 12.

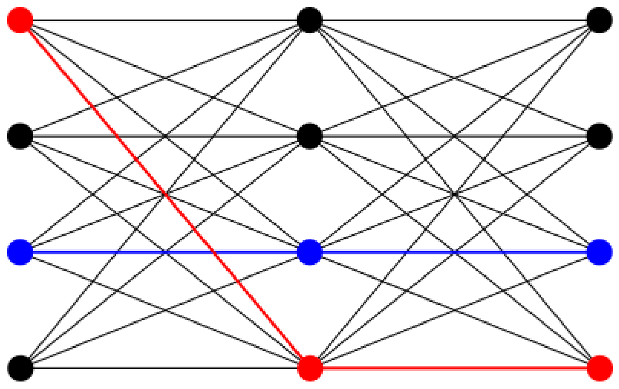


Figure 12: A cross-section that shows the descending red aircraft conflicting with the level blue aircraft

The solution that I developed was to make the agents reserve all the nodes between the altitude levels of the current and next node, when they are climbing or descending.

This gives the agent the horizontal and vertical protection when changing altitude. Figure 13 shows the nodes that would be reserved with a red border.

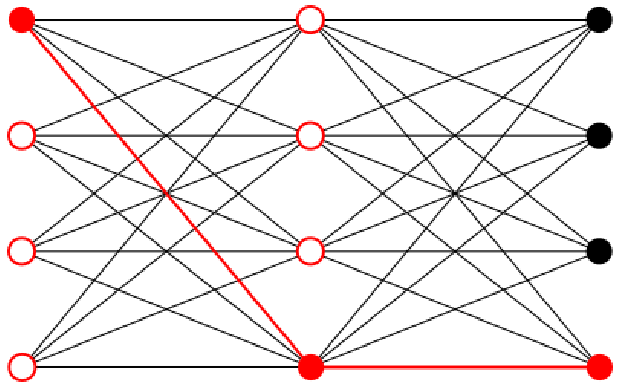


Figure 13: A cross-section with the block altitude node reservations marked

In order for this to be implemented, a reference to the node above and the node below had to be added to the *SpatialNode* class. When WHCA* is making node reservations, it will identify if there has been a change in altitude and then recursively follow the above/below node references to select the extra nodes which need to be reserved.

6.7 A* Timeout

So far, my implementation of A* has deadlock detection that will throw an exception if the search enters an infinite loop. However, under certain conditions, it is possible for the algorithm to take a large amount of time to complete. This is often due to having a very large graph or too many agents for the particular machine to handle. The result is, the algorithm cannot compute paths quick enough to keep up with the speed of the aircraft. The obvious solution is to use a computer with a faster processor and faster memory, but what if some unforeseen circumstances exceed the performance limit? There needs to be a robust, fail-safe mechanism to catch these cases.

The solution I used was to add a timeout mechanism to the search algorithm, I found that a five second timeout worked well. If a search exceeds the timeout limit, it will throw an exception that the agent manager will catch. It then handles it in the same way it would a deadlock: keep the agent where it is and reserve it's current node.

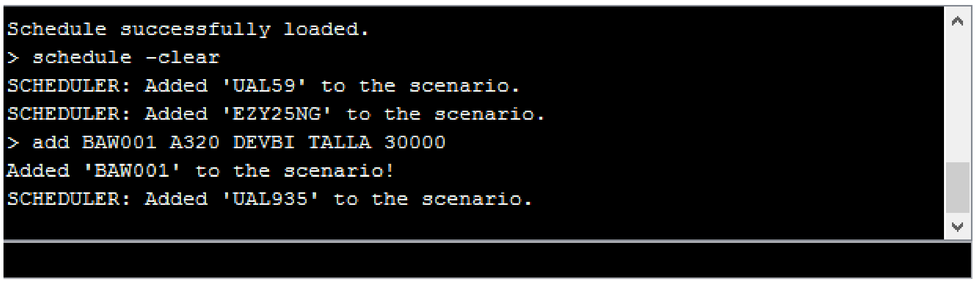
This strategy does not offer a complete solution to the problem, if there are too many aircraft then the vast majority will end up circling in the same position for a long time. But at the very least it offers a level of damage protection to prevent collisions.

6.8 User Interface

The user interface comprises of several key parts. The simulator provides the radar display for aircraft, as well as some scenario controls and statistics. The specification states that the AI should have a “form to manually add aircraft”, however, I decided that this would not be a valuable use of time. Instead, I implemented a very simple command line interface (Figure 14) that provides full functionality to control the AI, including;

adding/removing aircraft, loading flight schedules, enabling flight logging, clearing aircraft and the ability to query scenario statistics.

I also added Java command line options to allow scenarios to be ran in batch. It is possible to launch an instance of the program that will automatically load a schedule, start logging and close once the schedule has completed.



```
Schedule successfully loaded.
> schedule -clear
SCHEDULER: Added 'UAL59' to the scenario.
SCHEDULER: Added 'EZY25NG' to the scenario.
> add BAW001 A320 DEVBI TALLA 30000
Added 'BAW001' to the scenario!
SCHEDULER: Added 'UAL935' to the scenario.
```

Figure 14: Command line user interface

7 Evaluation

This section evaluates the effectiveness of various different parameters and compares the overall performance of the system to human air traffic controllers.

7.1 Heuristics

A heuristic is used by the path finding algorithm to guide the direction of search by measuring how likely a node is to lead to the goal node. The heuristic that the path finding algorithm uses has a significant impact on the performance and length of the paths found.

For a space-time problem, Silver [14] suggests that using true distance, the length of the shortest path through space ignoring any other agents, will be effective. Another possible heuristic is the Manhattan distance which is the sum of the distance between the X coordinates and the Y coordinates, similar to how one might navigate a city block. I decided to test three different heuristics: true distance, Manhattan distance and straight line distance. Figure 15 shows graphically how each heuristic is calculated.

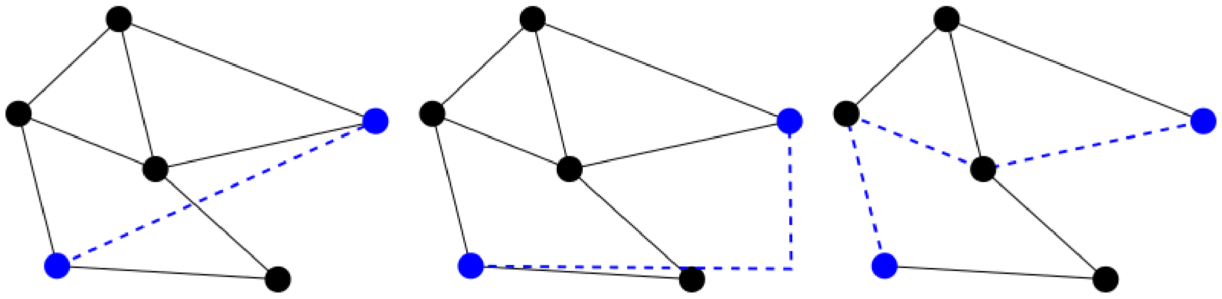


Figure 15: Left-to-right: straight line distance, Manhattan distance, true distance heuristics. The length of the dashed blue line represents the heuristic value.

The straight line distance heuristic simply uses the *estimateFlightTime()* function directly as its estimate. The Manhattan distance uses the function twice, once for the change in latitude and again for a change in longitude, the results are summed together. True distance uses the Floyd-Warshall algorithm discussed earlier in §6.5.

Experimentation Results

I tested the performance of each heuristic using three different flight schedules with 25 aircraft in the space of an hour. This is a similar volume of traffic that this area (Scotland) would see during peak times. To compare the results I calculated the optimality gap of each route for each aircraft:

$$optimality = \frac{actual\,route\,distance}{straight\,line\,distance} - 1$$

An optimality of zero is the most efficient route possible (i.e. a direct route). The mean optimality of each heuristic across the three scenarios can be seen in Figure 16. The graph shows that Manhattan distance leads to the least optimal routes, straight line distance performs slightly better and true distance is even better.

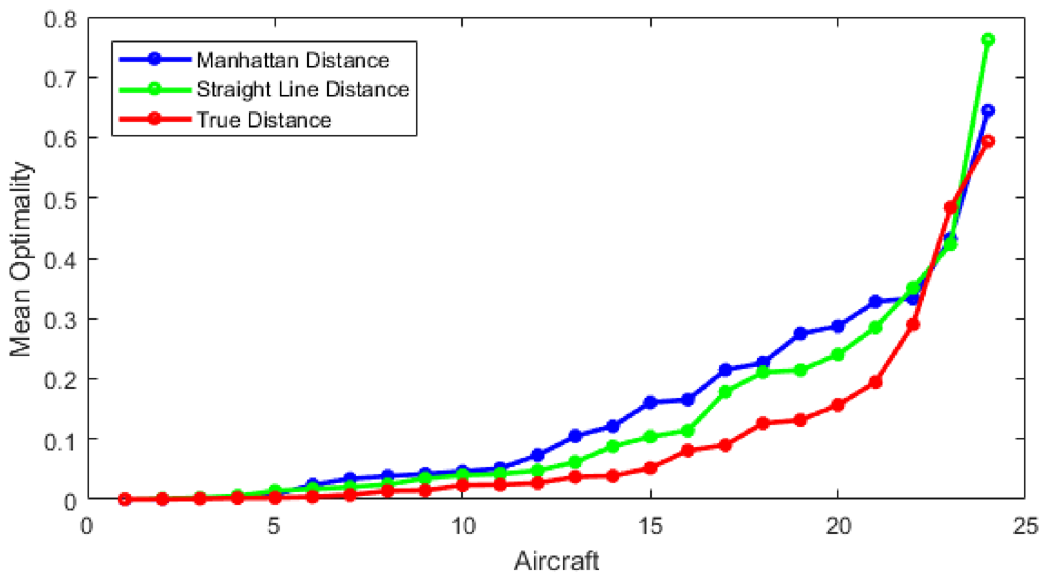


Figure 16: Mean optimality for three different heuristics

This trend can be seen clearly in Figure 17 which shows the routes for the same aircraft but using different heuristics. The true distance heuristic delivers an almost perfect route. Figure 18 shows the changes in altitude for this aircraft, it is clear that the Manhattan distance heuristic causes aircraft to constantly descend and climb, which is undesirable.

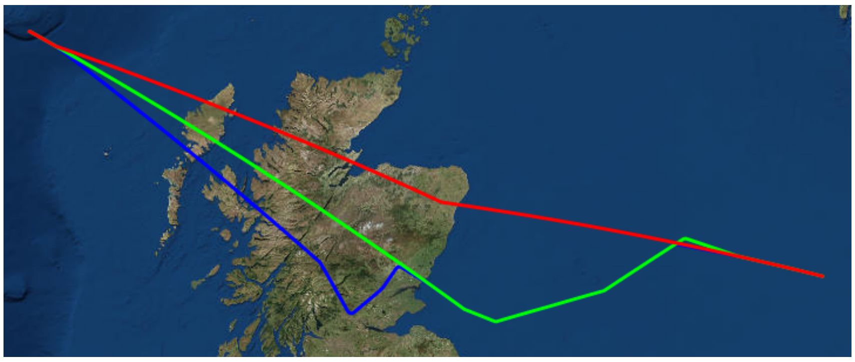


Figure 17: Aircraft track for each heuristic. Red - true distance, green - straight line distance, blue - Manhattan distance. *Satellite imagery courtesy of ESRI (http://www.esri.com/data/imagery).*

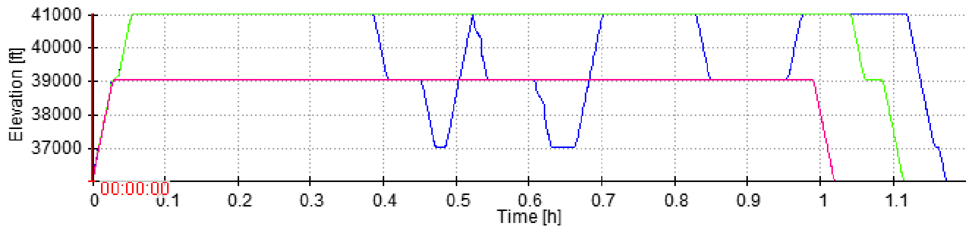


Figure 18: Aircraft altitude for each heuristic. Red - true distance, green - straight line distance, blue - Manhattan distance.

7.2 Agent Priority Schemes

All the agents in WHCA* have a priority value that dictates what order the agents will get to reserve their paths on the reservation table. The first agent will get full freedom to select the best path without any consideration for other agents, subsequent agents will have to avoid nodes reserved by previous agents. The order has a big impact on the result of the algorithm, on one hand you want agents to have an equal opportunity to be the first agent to select its path. However, you also don't want there to be too many changes of priority because it could lead to an unstable scenario with multiple aircraft in a deadlock where the priorities are constantly alternating and neither agent can get to its destination.

I tested four different scheduling disciplines to see which one would lead to the most optimal routes while avoiding deadlock scenarios. They included first-come first-serve (FCFS), furthest distance to goal, shortest job first (SJF) and agent agitation.

First-come first-serve

First-come first-serve behaves like a queue, where the first item in the queue is served before serving the item that is next in the queue [22]. In my system this is based on the airborne time of the aircraft. The aircraft which has been in the air the longest will have the highest priority and the aircraft which has been airborne for the shortest duration will have the lowest priority. The idea being that this will prevent agents from giving way to other agents indefinitely because over time its priority will increase, thus deadlocks should not occur.

Shortest job first

Shortest job first prioritises the least costing job first, the next least costing job second and so on. By prioritising shorter jobs first, it is good at maximising throughput [23]. My implementation measures each agent's distance to their goal and orders them starting with the closest. This strategy has the advantage of getting the aircraft on shorter routes out the way quickly, thus increasing the overall flow-rate.

Furthest distance to goal

Furthest distance to goal does the reverse to SJF and gives agents that are furthest away from their goal a higher priority. This means that over time an agent's priority will decrease. In theory this strategy should work well for maximising overall route optimality because short routes are often very close to been fully optimal as there are less alternative options for an aircraft, however, aircraft on shorter routes risk getting stuck in a deadlock situation where they constantly give way to the long-haul aircraft.

Agent agitation

Finally, Silver describes the concept of an 'agent agitation level' which represents how many times an agent has had to re-route and give way to other agents [12]. I decided to try prioritising agents using a similar metric by measuring how good the path so far has been, compared to the 'ideal' path. It uses the formula:

$$agitation = \frac{actual\, path\, cost}{heuristic\, path\, cost}$$

The idea is that agents which give way and inconvenience themselves will be rewarded by getting a higher priority in the next WHCA* window.

Experimentation Results

In order to compare the effectiveness of the four priority systems, I tested each system on three 25 aircraft/hour schedules in the Scottish region, this is the normal level of traffic during peak times for this area. Figure 19 shows how the priorities of aircraft change over time for one of the samples, a lower priority number represents a higher priority. Figure 20 shows a box plot of the optimality values for the three samples.

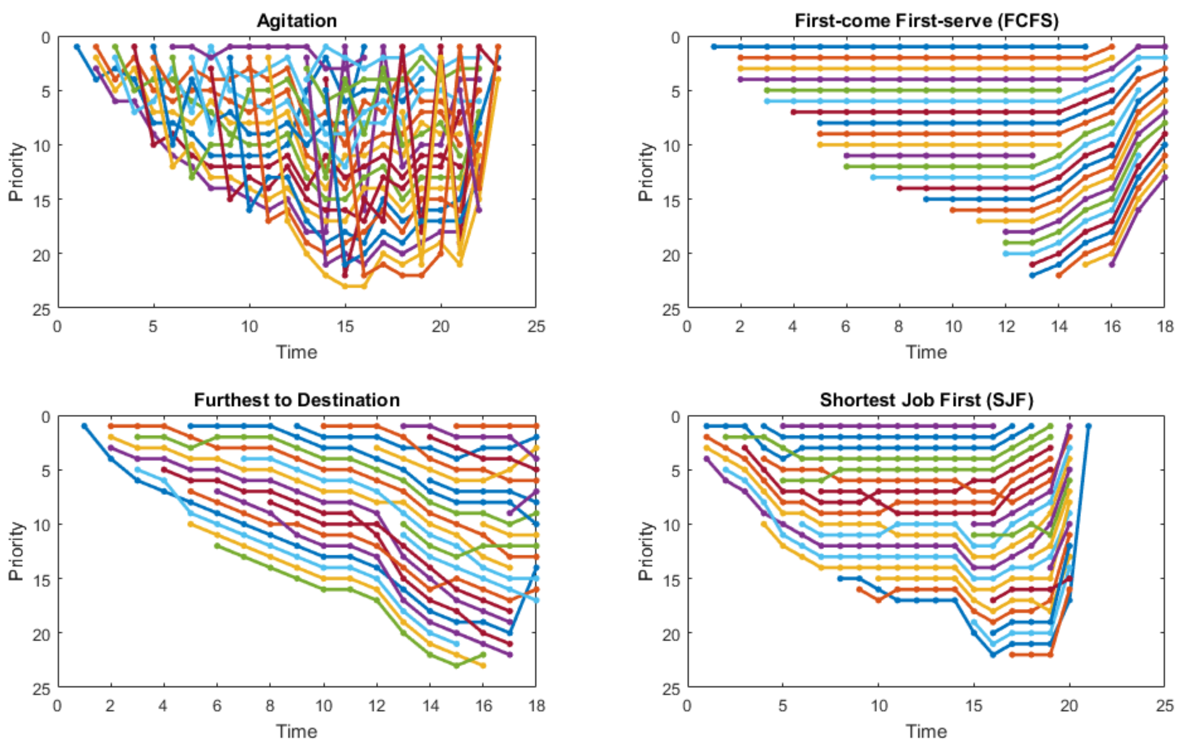


Figure 19: Agent priority diagrams for 4 different priority systems

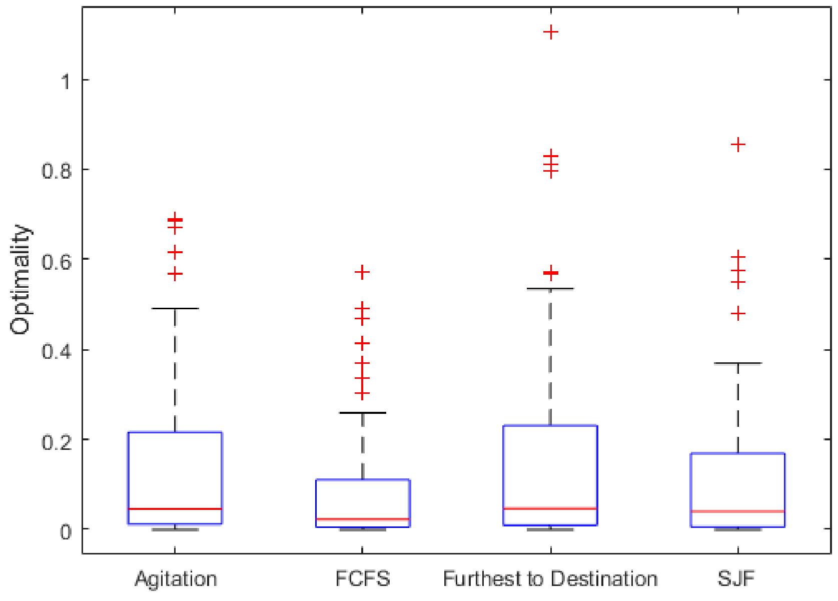


Figure 20: A box plot comparing the optimality of the 4 priority systems

It's quite clear to see that the agent agitation priority system is regularly changing agent priorities every iteration of WHCA*, in fact, a lot of the agents change from a very high to very low priority on almost every iteration. This puts the system in a dangerous situation where deadlocks could be very likely to happen, and, through my experimentation, this is the case. The final two aircraft in sample 1 shared the same goal node and were in similar positions, the aircraft would get about half way towards the goal before reaching the end of the WHCA* window. They would then swap priorities and recalculate their routes, but one aircraft would have to give way to the other – this cycle repeated indefinitely. Increasing the window size solved this particular case, however, it highlights that agitation is not a suitable priority scheme to use.

Furthest distance to goal seems to yield the least optimal routes. Figure 20 shows that this priority scheme has the largest range of optimality values as well as the least optimal mean value. This is most likely due to the fact that a large number of the scheduled flights have short routes to begin with and so automatically start with a low priority which only decreases over time. The shorter routes have less options for alternative paths and so if a required node becomes blocked, the diversion can lead to a drastically reduced optimality.

Shortest job first (SJF) performs slightly better than furthest distance to goal because it helps to solve the problem that furthest distance to goal has; where agents which are near to their goal have their route blocked and have to take an extremely inefficient diversion (often requiring backtracking). Agents further away from their goal have much more flexibility to make changes to their route which will have less impact on their route optimality. Figure 21 shows a comparison of the aircraft routes for both furthest distance and shortest job first. It's clear that shortest job first results in less backtracking and circling of aircraft which are near to their destination.

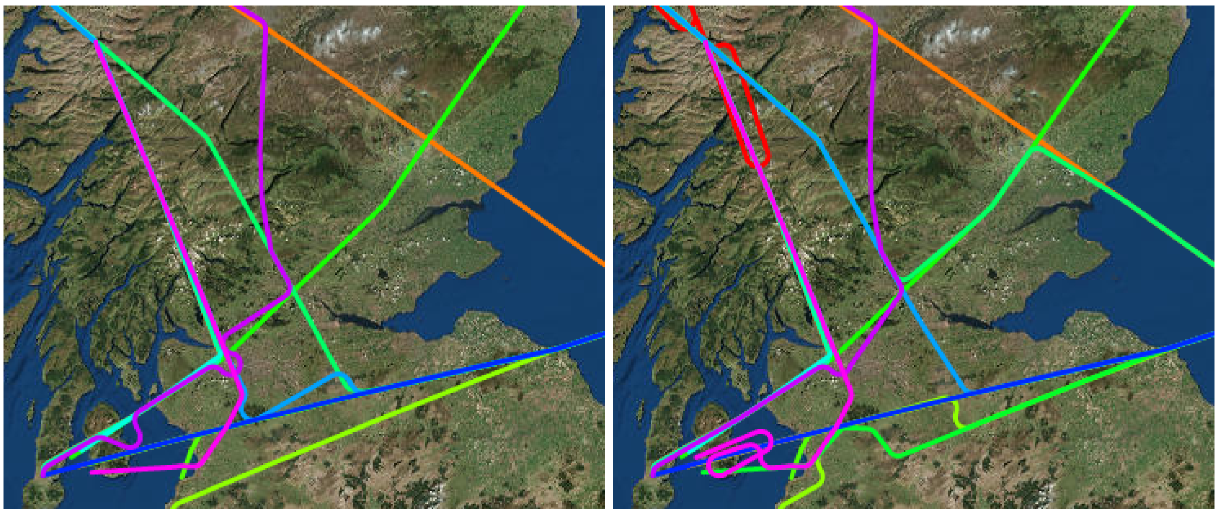


Figure 21: Route comparison for SJF (left) and furthest distance (right) priority systems. *Satellite imagery courtesy of ESRI (http://www.esri.com/data/imagery).*

The best performing priority scheme is first-come first-serve which gave the best mean optimality as well as the smallest range of optimality values. It's the simplest priority system available but seems to perform well and is not susceptible to deadlocks because there is no alternating of priorities between agents.

7.3 Agent Node Reservation Time

The amount of time that a node is reserved for by an agent is known as the node reservation time. In a simple multi-agent pathfinding scenario, agents will only need to reserve the node for one unit of time – the time that it is at the node. Aircraft separation is more complex because they need at least 5 nautical miles of lateral separation [4], however, an aircraft agent can be at any position along an edge, not necessarily at a node, so they need to reserve multiple nodes. The solution is to reserve nodes for t amount of time, where t is the time before the next agent can use the node.

Experimentation Results

I tested the impact of changing the amount of time that agents reserve nodes for, for four amounts of time: 10, 30, 60 and 90 minutes. In aviation, 10 minutes is the standard amount of separation time between aircraft [4], however, it should not be assumed that this will always lead to the most optimal routes.

Figure 22 shows the optimality of two different flight schedule samples for the range of four reservation times. The results for both samples show that a 10 minute reservation time is most optimal and that the larger the reservation time, the less optimal the route.

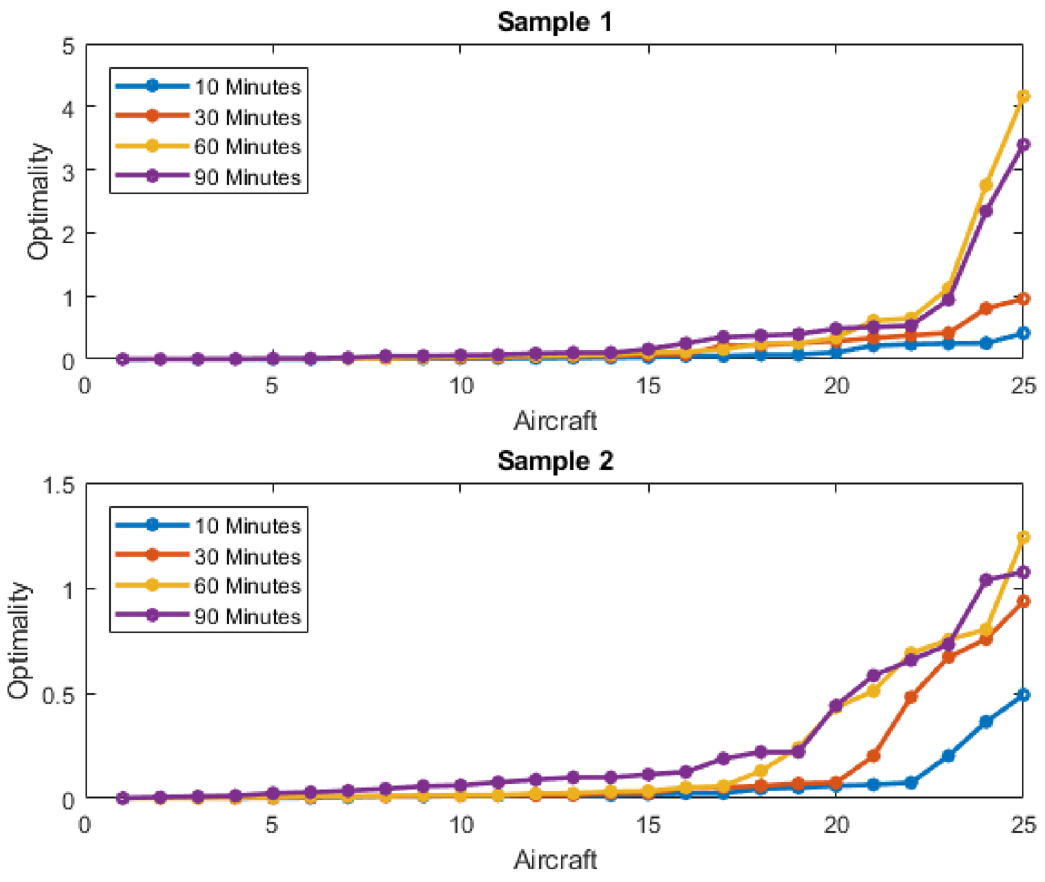


Figure 22: Optimality graphs for different node reservation times

There was also a noticeable increase in the number of deadlocks and timeouts occurring when using the larger (60 and 90 minute) reservation times. A lot of these timeouts also lead to conflicts between aircraft. This is due to the timed-out agents reserving their current location for large amounts of time, making the node impassable, which leads to further timeouts. This cascading cycle of timeouts resulted in multiple aircraft conflicting.

7.4 WHCA* Window Size

WHCA* will perform multi-agent pathfinding within a 'window' time period, once the agents reach the end of the window (or a specific point, known as K), the algorithm will calculate paths for the next window [12]. "The size of the window has a significant effect on the success and performance of the algorithm. With a large window, WHCA* behaves more like HCA* and the initialisation time increases. If the window is small, WHCA* behaves more like Local Repair A*. The success rate goes down and the path length increases." [12] Therefore, it is wise to select a window size which finds a balance.

I tried varying the size of the window for two different flight schedule samples. I used window sizes of 10, 30, 60, 90 and 120 minutes.

The optimality of the different window sizes had very little variation for both samples. I hypothesised that this was due to the fact that there is very little uncertainty in the simulator; all the aircraft behave very predictably, so there is no benefit to having a smaller window. I tried re-running the tests but every two minutes the wind direction and speed changes, thus introducing some uncertainty and unpredictability to the aircraft agents. I expected the smaller window sizes to perform best, but there was still little variation in the optimality.

I tried using different schedule samples and the results still showed little difference between the different window sizes. It is more than likely that randomly changing the wind direction is not enough because it effects all the aircraft equally. A better test would be to try introducing some subtle randomisation to the behaviour of aircraft, such as flying at a slightly faster/slower speed than expected or climbing/descending at a faster/slower rate. This would make aircraft behave much more unpredictably, which may make a smaller window size a desirable thing to have. I believe this should be a focus of any future work.

7.5 Intermediate Node Spacing

In §5.4, I describe the benefits of splitting long edges into smaller edges connected by intermediate nodes – essentially allowing more aircraft to be able to use the same airway simultaneously. The intermediate node spacing can be set to be very small, which will result in lots of intermediate nodes, or very large, which will have fewer nodes. I tested the effect of varying the spacing distance for 10, 20, 40 and 80 nautical miles.

Figure 23 shows that the total number of nodes in the graph decreases exponentially with the increase in intermediate node spacing. Therefore, for the sake of performance, it is best to select a spacing which achieves a high route optimality but does not negatively impact runtime performance by having too many nodes.

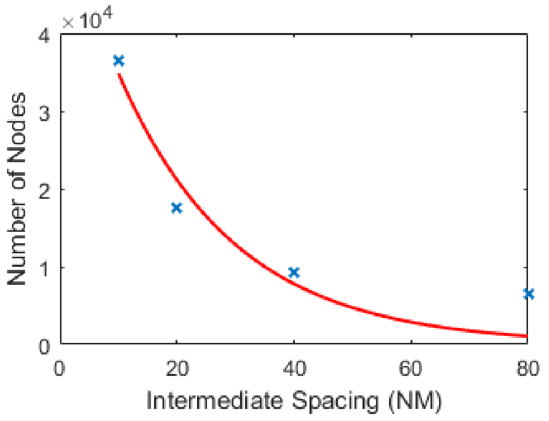


Figure 23: The number of nodes against the intermediate node spacing

I tested the different spacing distances on three different flight schedule samples, Figure 24 shows the average optimality for each. 10NM spacing has been excluded because the large number of nodes caused the program to run out of memory. There is not much variation between the optimality, 40NM performs slightly better, but I cannot conclusively say that it is the best spacing to use without further testing.

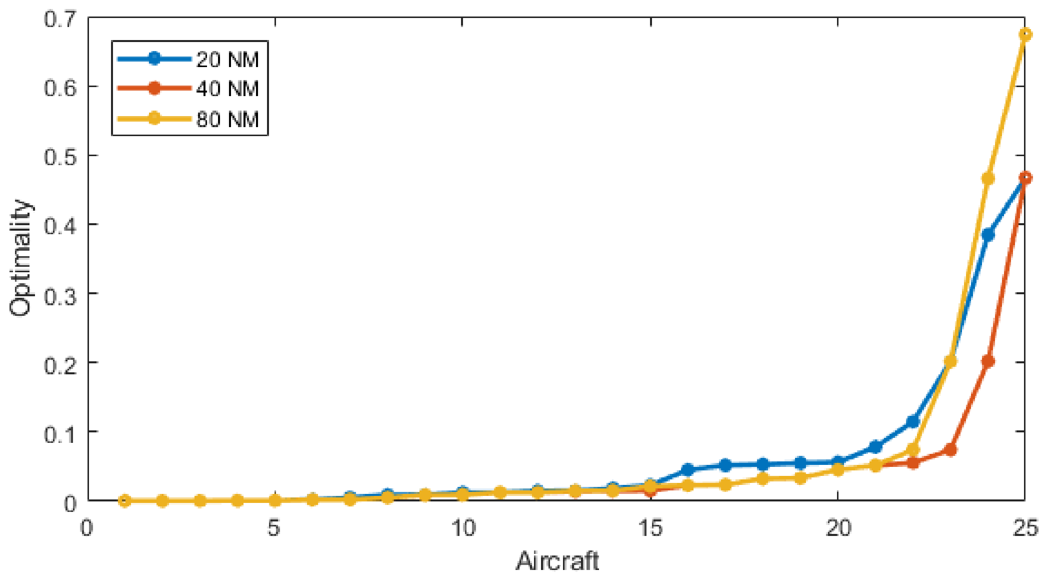


Figure 24: Optimality graph for different intermediate node spacing

The results indicate that 40NM might be the best spacing which achieves the balance of route optimality to a reasonable total number of nodes. Future research should look at testing this further.

7.6 Comparison against Human Air Traffic Controllers

The original aim of the dissertation was to use multi-agent path finding techniques to automate the control of en-route air traffic. In order to be accepted as a realistic alternative to human air traffic controllers, it needs to be proven that the system can safely handle the same quantity of traffic a human can and also direct aircraft on equally or more optimal routes.

Collecting Data

I used data from a website called ADS-B Exchange, which collates ADS-B data from receivers around the world [24]. ADS-B is a technology that aircraft use to broadcast their position, the broadcasts can be received by anyone with a receiving device that is in range [25]. The data comes in a raw CSV format which I processed to generate a GPX track file for each aircraft and a flight schedule which can be loaded into the AI. I did this for two, two hour periods of time.

Experiment

The flight schedules which I generated specify the start and goal locations. These locations do not necessarily match exactly to the ADS-B data, because, some of the time, ATC allows aircraft to route directly without following any airway. This means I have to start and end the aircraft at the nearest node to the actual start and end locations, which are normally within 10 nautical miles. Therefore, the route distances for the aircraft of the AI and real world cannot be compared directly. However, the route optimality can still be compared because the aircraft are still travelling in the same direction, for a similar distance and will need to de-conflict with the same aircraft.

Grid Airways

As an extra experiment, I decided to try running the scenarios on a grid-style graph, as well as the graph of existing airways. Figure 25 shows the model I used, which has a node for every degree of latitude and longitude. My hypothesis was that the grid will allow agents to find more direct routes to their goal nodes than the existing airways currently allow.

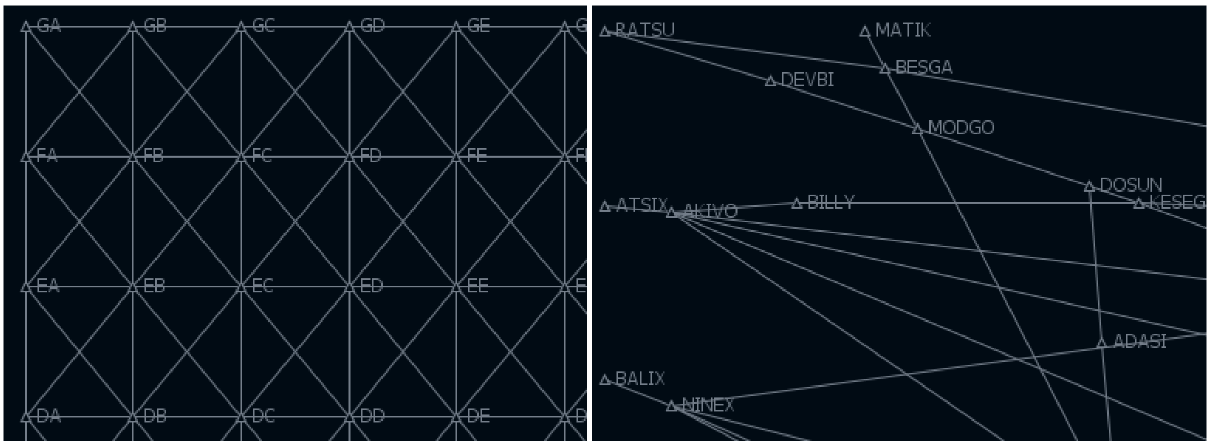


Figure 25: Grid-based model (left) compared to the existing airways (right)

Results

I generated two statistics to use for comparison. Firstly, the route optimality, as used in previous experiments. Secondly, the 'average speed', which is calculated from the route straight line distance and the time it took the agent to reach the goal. The higher the average speed, the better the route is. Figure 26 shows the optimality for both schedule samples and Figure 27 shows the average speed, displayed as a boxplot.

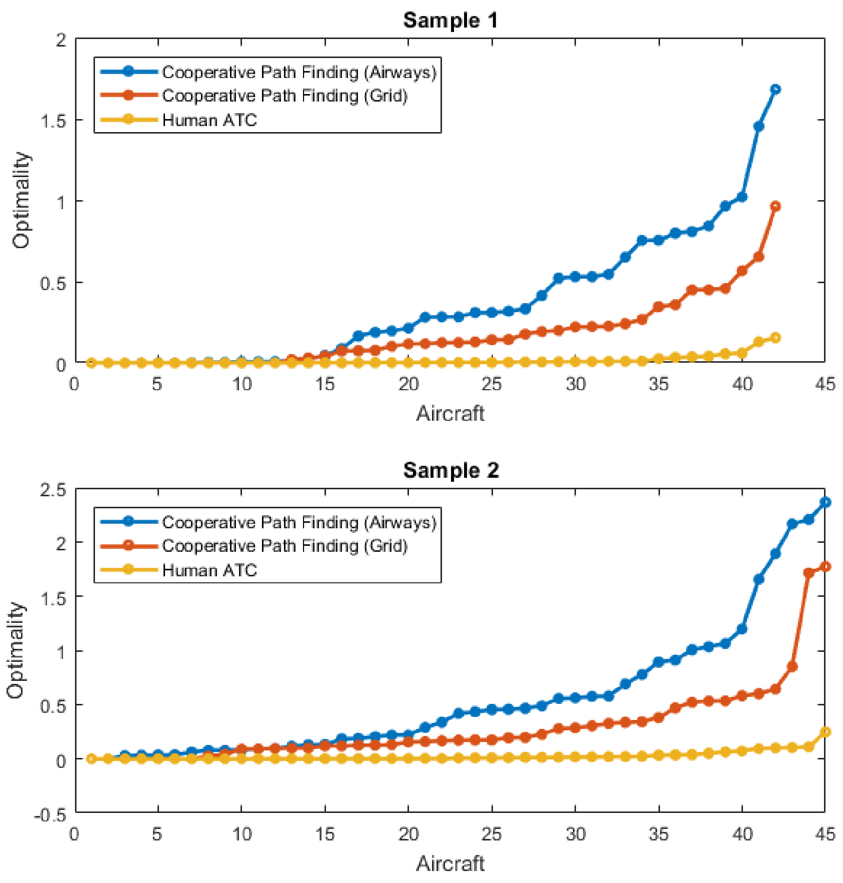


Figure 26: Optimality graph comparing MAPF to human ATC

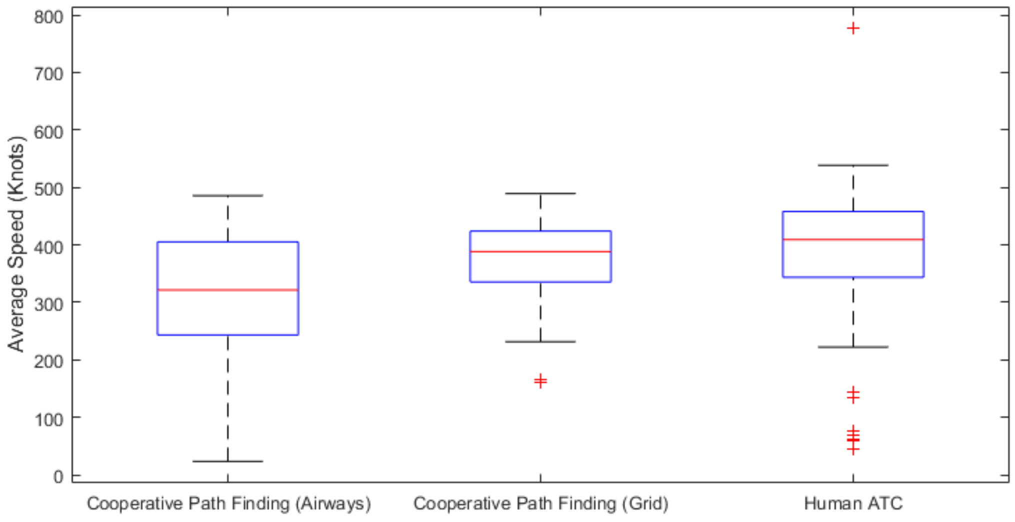


Figure 27: Average aircraft speed comparison for MAPF and human ATC

From the results, I can conclude that the human air traffic controller performed better in terms of route optimality and in terms of average aircraft speed. Included in Appendix

B are satellite maps which depict the routes taken by aircraft controlled by humans and the AI. It is clear that the vast majority of human controlled aircraft follow direct routes to their destination, which is impossible for the AI to do, unless there happens to be an existing airway there. This is an underlying limitation to using a graph based approach, which will be discussed further in the summary and reflection section.

The grid based cooperative path finding strategy outperformed the existing airways system because the optimality is lower and average speeds are higher. In fact, the range of average speeds for the grid system are smaller than what human air traffic controllers achieve, which indicates that overall it provides more consistent routes.

The system did successfully manage to handle the quantity of aircraft that the Scottish flight region would see during a busy period. I also tested it with a higher quantity of aircraft (35 within one hour), and it did successfully handle this density of traffic. This shows that, although it may not be able to deliver the most optimal routes, MAPF is a robust approach to handling high quantities of traffic safely.

8 Summary and Reflections

This section reflects on the success of the project in terms of meeting requirements and project management. Further improvements and developments are also proposed.

8.1 Project Management

Throughout the project I kept about two weeks ahead of schedule because I was able to gain time early on. Rather than slowing down the work rate to follow the schedule, I decided to shift the schedule forwards and use the two weeks gained as extra contingency time. This extra time proved to be very helpful because I was able to spend more time on tasks K and L, which I believe massively benefited the project. If I was to plan this project again, I would allow less time for the planning stage and instead focus more of the available time on the elements which have a lot of uncertainty regarding the amount of time they will take, such as the MAPF module. The original time plan and Gantt chart are included below:

A Write the project proposal. *(1.5 weeks)*

B Research the constraints and objectives used in modern air traffic control. *(0.5 weeks)*

C Research what, if any, AI tools are already used in air traffic control. *(0.5 weeks)*

D Research existing multi-agent path finding techniques. *(0.5 weeks)*

E Design the flight simulator component. *(1 week)*

F Develop the flight simulator. *(3.5 weeks)*

G Write the interim report. *(1.5 weeks)*

H Model the airways for a particular region in a data structure that can be used by the MAPF techniques researched. *(2 weeks)*

I Implement an MAPF technique to control aircraft in the flight simulator. *(4 weeks)*

J Implement deadlock detection and recovery functionality. *(1 week)*

K Experiment with different parameters and methods to achieve the most reliable separation of aircraft and the most efficient use of airspace. *(1 week)*

L Analyse and evaluate results of the MAPF technique used. *(1 week)*

M Finalise dissertation. *(2 weeks)*

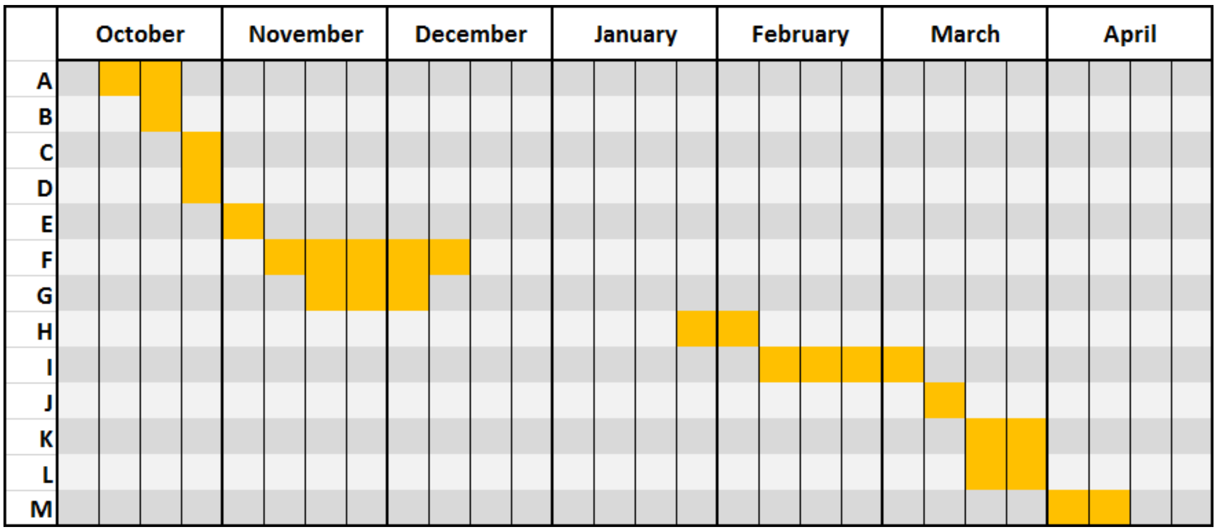


Figure 28: Original time plan from the project proposal

I tried to complete tasks in an agile scrum fashion, where each individual task makes up a sprint (except the simulator and MAPF development, which was broken down into smaller sprints). I decided to swap tasks H and I because I felt that it would make more sense to implement a generic MAPF technique before trying to model the airways as a graph or control aircraft. The breakdown of sprints are included in Appendix C.

8.2 Project Appraisal

If I was to start this project again, I would spend more time planning what and how I was going to compare the different parameters of the search algorithm. For example, I didn't consider the need for a flight logger component until mid-way through the implementation. If I had decided how I was going to record the data and what I was going to compare earlier on, I may have being able to draw more conclusions about the system. However, I'm pleased I made the decision to implement the logger, even if it wasn't included in the original specification.

I think that my time management and distribution of work was good. The original time plan overestimated most tasks, but this proved to be a good thing and I'm pleased that I overestimated instead of underestimated.

Overall, I believe the project has been successful. The project has proven that MAPF is potentially a suitable approach to automating the en-route control of aircraft. Although, in its current state, it struggles to match the optimality of a human air traffic controller, it can still provide a level of safe separation for air traffic. This is obviously the most important factor for any safety critical AI.

The problem of inefficient routes is unfortunately an inherent limitation of using a graph model. Agents are limited to following edges and nodes in the graph, preventing any opportunity to 'cut corners' and fly direct to their goals. It is also very limiting if aircraft need to make diversions for emergencies or bad weather, the aircraft may need to backtrack in order to follow a different path.

The grid based approach that I experimented with in §7.6 shows much more promising results because aircraft are able to follow direct routes much more easily. In fact, the whole concept of airways is very limiting; NATS has acknowledged this by beginning to introduce a system called Direct Route Airspace (DRA) which gives aircraft and controllers the freedom to route aircraft on the most optimal routes based on the conditions of the day, without the limitation of airways [26]. If this system was to be more widely adopted, a grid based system, like the one I tested, could be a realistic way to automate the control of aircraft inside the airspace.

In its current state, aircraft do occasionally conflict with each other. The vast majority of conflicts happen when a new agent is added to the system in the same location, or near to, an existing agent. The system does not have enough time to avoid the inevitable conflict. This is simply down to the implementation of the scheduler, so I did not spend too much time trying to mitigate this. A realistic solution would be to prevent new agents from being added if the surrounding area is occupied. This highlights that there will need to be careful consideration to how the system would integrate with surrounding automated or un-automated systems.

8.3 Future Work and Conclusion

As identified in the appraisal section, the main caveat of the MAPF system is the limitation imposed by the graph model. Although MAPF has proven itself to be a reliable way of separating aircraft, it falls down on finding optimal routes. An area which would be worthwhile researching further is the type of graph used and how aircraft use it. For example, one possible idea is to divide the airspace into small cubes of space (perhaps 5 nautical miles wide by 1000ft high) and use MAPF algorithms to reserve paths through the voxel grid. The routes can be as direct as possible but the reservations are made for the voxels that the route intersects.

One of the fears in the industry currently is that artificial intelligence is like a "faceless 'black box'" [27] that can behave unpredictably and few people can understand. Technologies such as genetic algorithms and neural networks are guilty of this; they can often be quite unpredictable. Path finding algorithms, on the other hand, are much simpler to understand. They behave consistently and reliably, so people will be more accepting of them. For safety critical systems, like air traffic control, this is incredibly important. The results of this dissertation show quite clearly that MAPF is a feasible alternative to 'black box' style AI and that it is a strategy that can deliver a reliable and safe outcome.

Glossary

Agent	An independent entity that navigates through the graph [12]. In this project, the agents are the aircraft.
Air Traffic Control (ATC)	Air traffic controllers are responsible for directing aircraft to their destination, while avoiding conflicts with other aircraft. Controllers work together to provide total coverage for all the airspace [4].
Airspace	Any area which an aircraft flies through. Can be controlled (by ATC) or uncontrolled – where aircraft are free to do what they want [4].
Airways	A network of corridors of controlled airspace which "are normally 10 miles wide and extend from a few thousand feet above ground to 24,500ft" [4].
Conflict	When two or more aircraft being controlled by ATC come within 5 nautical miles horizontally and less than 1000ft vertically, a conflict is said to have occurred. This is the standard level of separation ATC aims to achieve, anything less would be considered dangerous [4].
Heading	The direction that an aircraft is pointing in [28]. A heading can be assigned to an aircraft by an air traffic controller.
Instruction	An air traffic controller can instruct an aircraft under their control to do certain things; such as climb, descend, turn or fly to a waypoint.
NATS	Formerly National Air Traffic Services, provide air traffic control services for the majority of the United Kingdom, which is the focus area of this project.
Nautical Mile (NM)	A unit of distance used in aviation; equivalent to 1852 metres.
Track	The actual direction that an aircraft moves in, a combination of the heading/speed and wind direction/speed vectors [28].

Bibliography

[1] NATS, “Our history.” [Online]. Available: https://www.nats.aero/about-us/our-history/

[2] T. Rutherford, “Air transport statistics,” House Of Commons Library, 2011.

[3] A. F. Libraries, “History of air traffic control.” [Online]. Available: http://www.actforlibraries.org/history-of-air-traffic-control/

[4] D. Graves, *UK Air Traffic Control: A Layman’s Guide*, 1998.

[5] “Flightradar24.” [Online]. Available: https://www.flightradar24.com/

[6] D. Delahaye and S. Puechmorel, *Modeling and Optimization of Air Traffic*, 2013, pp. 85-87.

[7] M. Rolfe, “How technology is transforming air traffic management,” 2013. [Online]. Available: https://nats.aero/blog/2013/07/how-technology-is-transforming-air-traffic-management/

[8] NATS, “Congestion.” [Online]. Available: https://www.nats.aero/services/challenges/congestion/

[9] J. O. T. Lehouillier, F. Soumis and C. Allignol, “Measuring the interactions between air traffic control and flow management using a simulation-based framework,” *Computers & Industrial Engineering*, vol. 99, pp. 269–279, 2016.

[10] S. Corver and G. Grote, “Uncertainty management in enroute air traffic control: a field study exploring controller strategies and requirements for automation,” *Cognition, Technology & Work*, vol. 18, pp. 541–565, 2016.

[11] J.-M. A. N. Durand and J. Noailles, “Automatic aircraft conflict resolution using genetic algorithms,” *ACM symposium on Applied Computing*, 1996.

[12] D. Silver, “Cooperative pathfinding,” *1st Artificial Intelligence and Interactive Digital Entertainment Conference*, 2005.

[13] Z. Bnaya and A. Felner, “Conflict-oriented windowed hierarchical cooperative a*,” in *IEEE International Conference on Robotics & Automation (ICRA)*, 2014.

[14] D. Silver, “Cooperative pathfinding,” in *AI Game Programming Wisdom 3*, 2006, pp. 99–111.

[15] S. J. Russell and P. Norvig, *Artificial Intelligence A Modern Approach*, 3rd ed. Pearson, 2009.

[16] T. L. Turocy and B. von Stengel, “Game theory,” Oct. 2001, cDAM Research Report LSE-CDAM-2001-09.

[17] J. Ross, “Openscope air traffic control simulator,” 2017. [Online]. Available: https://github.com/openscope/openscope

[18] T. H. Cormen, *Introduction to Algorithms*, 3rd ed. Cambridge, Mass. ; London : MIT Press, 2009.

[19] "Iso/iec/ieee international standard - systems and software engineering – vocabulary," *ISO/IEC/IEEE 24765:2010(E)*, pp. 1–418, Dec 2010.

[20] FlightLearnings, "Airspeed (part two)," 2017. [Online]. Available: http://www.flightlearnings.com/2017/08/03/airspeed-part-two/

[21] C. Veness, "Calculate distance, bearing and more between latitude/longitude points," 2017. [Online]. Available: https://www.movable-type.co.uk/scripts/latlong.html

[22] A. J. R. Robert L. Kruse, *Data Structures and Program Design in C++*. Alan Apt, 2000.

[23] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 2015.

[24] "Ads-b exchange." [Online]. Available: https://www.adsbexchange.com/

[25] A. Australia, "How ads-b works." [Online]. Available: http://www.airservicesaustralia.com/projects/ads-b/how-ads-b-works/

[26] P. Beauchamp, "The journey to free route airspace," March 2015. [Online]. Available: https://nats.aero/blog/2015/03/journey-free-route-airspace/

[27] D. C. L. Cras, "Machine learning, artificial intelligence and air traffic management," February 2018. [Online]. Available: https://nats.aero/blog/2018/02/machine-learning-artificial-intelligence-air-traffic-management/

[28] FlightLearnings, "Effect of wind on navigation (part two)," 2018. [Online]. Available: http://www.flightlearnings.com/2012/07/16/effect-of-wind-on-navigation-part-two/