Terrain-aware Simulator for Evaluating Flight Plans for Unmanned Aerial Vehicles

Patrick Barry

Final Year Project
BSc in Computer Science

Supervisor: Cormac Sreenan
Second Reader: Kenneth Brown

Department of Computer Science
University College Cork

April 2023

Abstract

Unmanned aerial vehicle delivery, more commonly referred to as drone delivery is a rapidly growing field with many competitors including Google, Amazon, UPS and many more specializing in the field. Better fuel efficiency, quicker delivery journeys, and less road traffic are just some of the reasons this field is growing so quickly. The maximum altitude a drone can fly in most countries is 120 meters above the closest point of the earth's surface. Despite the drone being capable of flying over any obstacles such as mountains or buildings, it is often advised to find a path around these obstacles to minimise energy consumption. In this project, various methods are tested and implemented on a topographic map of Ireland to return a path that a UAV can follow while avoiding any physical obstacles. Using several metrics including energy consumption and distance covered, these methods are then compared given the same start and destination coordinates.

Declaration of Originality

In signing this declaration, you are conforming, in writing, that the submitted work is entirely your own original work, except where clearly attributed otherwise, and that it has not been submitted partly or wholly for any other educational award.

I hereby declare that:

- this is all my own work, unless clearly indicated otherwise, with full and proper accreditation;
- with respect to my own work: none of it has been submitted at any educational institution contributing in any way to an educational award;
- with respect to another's work: all text, diagrams, code, or ideas, whether verbatim, paraphrased or otherwise modified or adapted, have been duly attributed to the source in a scholarly manner, whether from books, papers, lecture notes or any other student's work, whether published or unpublished, electronically or in print.

Signed: P. Barry
Date: 24/04/2023

Table of Contents

Abstract **i**

Declaration of Originality **ii**

1 Introduction **1**

1.1 Objective 1

1.2 Overview of Project 1

1.3 Report Outline 2

2 Analysis **3**

2.1 Background 3

2.2 Delivery Drones in the Market 4

2.3 Related Work 5

2.3.1 Numerical Optimisation Algorithms 5

2.3.2 Bio-inspired Algorithms 6

2.3.3 Sampling-Based Algorithms 7

2.3.4 Graph-Search Based Algorithms 8

2.4 Research Gap 9

2.5 Project Requirements 10

2.5.1 Core Functional Requirements 10

2.5.2 Non-Core Functional Requirements 10

3 Design **11**

3.1 Topographic Map 11

3.2 Algorithm Designs 11

3.2.1 Obstacle Avoidance using Circles 11

3.2.2 Obstacle Avoidance using Waypoints within Circles . . 14

3.2.3 Follow the Edge 16

3.2.4 Greedy 18

3.2.5 A* 20

3.2.6 Bisect Obstacles 22

4 Implementation **24**

4.1 Programming Language Used 24

4.2 Accessing a Topographic Map 24

4.3 Accessing the Data in the Map 24

4.4 Data Structures and Classes 25

4.4.1 Line Class 25
4.4.2 Obstacles Class 26
4.4.3 Circles Class 27
4.4.4 Area Map Class 28
4.5 Algorithm Implementations and Challenges 29
4.5.1 Algorithms using Circles 29
4.5.2 Follow the Edge Algorithm 31
4.5.3 Greedy and A* Algorithms 34
4.5.4 Bisect Obstacles 35
4.6 Additional Methods Implemented 37
4.6.1 Generating an Obstacle Map as a Graph for A* 37
4.6.2 Path Smoothing 38

5 Evaluation **39**
5.1 Outline of Tests Performed 39
5.1.1 Metrics Used - Completeness 40
5.1.2 Metrics Used - Distance Covered 40
5.1.3 Metrics Used - Energy Consumption 40
5.1.4 Unit Conversion from Joules to Milliamp Hours 41
5.2 Optimal Flight Path Analysis 42
5.2.1 Path Returned - Turns 44
5.2.2 Paths Returned - Metres Ascending and Descending . . 45
5.2.3 Paths Returned - Distance Ratio 50
5.3 Requirements Satisfied 51

6 Conclusion **52**
6.1 Project Summary 52
6.2 Future Work 52
Bibliography

List of Figures

1 Example illustration of PRM Algorithm 8
2 A* Path on Voxel Space 9
3 Obstacle Avoidance using Circles Path Returned - Example . 13
4 Obstacle Avoidance using Waypoints within Circles Path Returned - Example 15
5 An example where the Follow the Edge algorithm is successful 17
6 Example of Path Returned by Follow the Edge 17
7 Example of Path Returned by Greedy 19
8 Example of Path Returned by A* 21
9 Illustration of Semi-Circle - Solution 1 29
10 Circles Algorithm Problem Illustration 30
11 Follow the Edge Illustration 32
12 Problem with Follow the Edge 33
13 Example of A* Graph 35
14 Bisect Obstacles Illustration 35
15 Path Smoothing Example from [11] 38
16 Two Bar Charts about Angles on Paths Returned 44
17 Elevation Change across Path - First Pair of Points 46
18 Elevation Change across Path - Second Pair of Points 47
19 Elevation Change across Path - Third Pair of Points 49
20 Distance Ratio Comparison 50

List of Tables

1 Results without Path Smoothing Applied 42
2 Results with Path Smoothing Applied 43
3 Requirements Satisfied 51

1 Introduction

1.1 Objective

Interest in using unmanned aerial vehicles (UAVs) as a method of last-mile delivery has grown rapidly in recent years [40], which involves using a UAV or a drone to transport small goods from a warehouse or distribution centre to their final destination. There are several advantages that UAVs offer, including quicker deliveries, reduced delivery costs and a lower carbon footprint for transportation [9, 16, 49]. However numerous obstacles stand before the drone delivery field, such as addressing concerns of safety and privacy [28], retrieving the necessary approval from government authorities [45], and the limitations of UAVs, such as battery capacity. Among these obstacles is finding an algorithm or a method to calculate the optimal flight path for the UAV taking wind speed and direction, energy consumption and safety into account.

This project takes two points as input, returning the path with minimal energy consumption. Various factors were considered to calculate the energy consumption, including the distance flown, the number of changes in direction or turns made, the angle of each of these turns and the number of metres spent climbing and descending. The objective of the project is to develop an algorithm which evaluates the terrain along with the other factors mentioned, and returns an optimal flight path for the UAV.

1.2 Overview of Project

Throughout this project, several algorithms were designed and implemented, including well-known path-finding algorithms *A** [17] and *Greedy* which were chosen to ensure firstly completeness and then optimality in the case of *A**. All implementation was done using Python due to its numerous packages which make accessing data from topographic maps much simpler. This report describes the *Obstacle Avoidance using Circles*, *Obstacle Avoidance using Waypoints within Circles*, *Follow the Edge*, and *Bisect Obstacles* algorithms in detail, discussing the strengths and weaknesses of each approach. The report also documents the implementation of *Greedy* and *A** algorithms and the results achieved. Also discussed are the challenges involved in accessing a topographic map and retrieving data from the map. The classes and data structures implemented to use with each algorithm are also described.

1.3 Report Outline

Section 2 analyses the problem faced in this project, giving some background on the field of drone delivery, summarising some existing work on the topic and stating the project requirements. Section 3 describes the topographic map used and details the algorithm designs with the use of illustrations and pseudo-code. Section 4 discusses how the topographic map was chosen and how the data from the map was accessed. details the implementation of the algorithms, including any technical challenges faced and how they were resolved, how the returned paths were analysed and tested, and the main data structures employed. Section 5 evaluates and compares all algorithms using graphical representations of the results and explaining the conclusions gathered from these results. Section 6 summarises the project, reflects on what has been achieved, and discusses any future developments that could be made.

2 Analysis

2.1 Background

Unmanned aerial vehicles (UAVs), specifically drones, have become a widely used technology, with over 800,000 drones registered in the US [54]. UAVs operate autonomously with no human involvement onboard piloting the aircraft [2]. Growing interest in autonomous drones has resulted in their use within agricultural, health, military, and meteorological domains [2]. Additionally, UAVs are becoming more commonplace in urban areas being used as a method of last-mile delivery, which involves transporting goods from a distribution center or warehouse to their final destination. Drone Industry Insights, a leading source for data on drones, published in 2019 the Drone Delivery Market Report 2019-2024 and predicted that drone delivery would be the top application for drones globally by 2024 [19]. This has not yet been proven true, with several other drone applications still much more common than drone delivery, including ”Mapping and Surveying”, ”Inspecting” and ”Photography and Filming” [20]. In recent years, the number of packages delivered using UAVs has, however, increased dramatically, with huge corporations Alphabet (Wing), UPS (Flight Forward) and Amazon (Prime Air) all spending millions of dollars on research, finding varying levels of success in their efforts [8, 57, 10].

Manna and Zipline are examples of companies specializing in the use of UAVs as a method of last-mile delivery. Manna are an Irish company who have made over 100,000 deliveries of packages up to 2 kilograms, generally small grocery items or medical supplies [13]. Zipline began operating in Rwanda in 2016 delivering blood supplies to local hospitals [5]. They have since completed over 500,000 deliveries and launched a new drone model in March 2023 which is better suited to urban areas, as the previous model required a large distribution centre for take-off and landing of the UAVs [34]. Both of these companies require a human operator to supervise flights, meaning they are not fully autonomous [21, 64].

There are many problems preventing this service from becoming the primary last-mile method, including limited payload, limited range, altitude regulations [4] and noise pollution as more UAVs enter the airspace [43].

The range of these UAVs is generally limited to 5-15 kilometres at a speed of around 80 km/h [36], while the maximum payload is typically around 3 kilograms. In most countries around the world, there is a maximum altitude of 120 metres above the closest point of the earth's surface [4]. This reduces the risk of encountering other aircraft on the path, with most activities occurring above 150 metres.

This project will focus on developing pathfinding algorithms for these drones to avoid any obstacles along the path while adhering to the strict height regulations.

2.2 Delivery Drones in the Market

There are many companies researching UAV technology as a method of last-mile delivery, including Amazon, Manna, and Zipline.

The CEO of Amazon, Jeff Bezos, predicted in 2013 that UAVs would be delivering packages to customers doorsteps within five years [6]. Ten years later and Amazon's Prime Air service is still not fully launched across the US [18]. There are several factors that have been blocking the progress Jeff Bezos expected, including receiving the necessary approval from the Federal Aviation Authority (FAA) [45], addressing privacy and safety concerns [28, 33], and developing reliable algorithms and methods to calculate flight paths for drones [55]. Safety and reliability issues have been present throughout the service's lifetime, with one crash leading to a 25 acre brush fire in the testing facility in Oregon [50]. This crash was caused as the drone attempted to change it's trajectory from flying upwards to maintaining it's altitude and the drone's motor shut off, resulting in it spiralling to the ground and igniting the brush fire [50].

Other companies have found more success in using UAVs as a method of last-mile delivery, including Manna, an Irish start-up spearheading the technology in Europe [42], and Zipline, an American company who have delivered over 500,000 packages [53]. Manna have tested their services in several different zones across Ireland, and are currently based in Balbriggan, Dublin. They focus on delivering small goods up to two kilograms, with a range of about two kilometres [36]. Zipline began operating in 2019, delivering blood supplies to nearby hospitals in Rwanda, and have since expanded their operations across the world, with deliveries now being made in Rwanda, Japan, Ghana, Nigeria, and the United States [47].

One of the biggest hurdles for Amazon was receiving approval from the FAA to operate their drone delivery service, Prime Air, across the United States. This approval was granted in 2020, but came with heavy restrictions, including flight over people being prohibited unless approved by an administrator and flight being prohibited if the drone is within 75 metres of a moving vehicle, with an Amazon employee having to stand by main roads to look for approaching vehicles [18].

2.3 Related Work

Path planning is defined as the process of finding a set of points for a vehicle, often autonomous, to follow from its source to a destination while avoiding collision with obstacles in the terrain [44]. Path planning is an essential component of drone deliveries, as finding the optimal path reduces costs, reduces delivery times, and increases the efficiency of each delivery [16]. There exists two types of path planning for UAVs, offline, where the entire path is calculated prior to take-off, and online, where the path is dynamically calculated as the UAV encounters unexpected or moving obstacles while flying [59].

There are numerous research papers on the topic of path planning in UAVs, with various different application areas for the UAVs, with a wide array of different algorithms implemented. These algorithms can be categorised as numerical optimisation algorithms, bio-inspired algorithms, sampling-based algorithms, and graph-search based algorithms [35].

2.3.1 Numerical Optimisation Algorithms

Numerical optimisation algorithms model the environment as well as the UAV specifications, considering the kinematic and dynamic constraints, and creating a cost function with these constraints in mind to find the optimal path. Some commonly used methods include Mixed-Integer Linear Programming [3] and Binary-Linear Programming [37].

Alotaibi et al. [3] is concerned with path planning of UAVs in a military application, and aims to maximise the number of targets visited for multiple UAVs, while limiting the threat exposure and the time spent traveling for each UAV to a constant pre-determined value. A number of waypoint-generation methods are investigated including the Maximin Threat Reduction method, the Rectangular Threat Reduction method and the Branch-and-Cut-and-Price method, with the latter found to be the most effective.

2.3.2 Bio-inspired Algorithms

Bio-inspired algorithms are methods to solve the path planning problem that are inspired by nature and the behaviour of living organisms. There are several examples of bio-inspired algorithms, the most prominent including Ant Colony Optimisation (ACO) [62], Genetic Algorithms (GA) [48], and Particle Swarm Optimisation (PSO) [44].

An ACO algorithm involves mimicking the behaviours of ant colonies in nature. Ants deposit pheromone as they move, which other ants can sense and follow. Ants are often looking for food, and hence the paths with the highest pheromone levels are the most likely to lead to food [15]. Applying this logic to the pathfinding problem, a number of ants can be initialised as agents in a search space. Each ant chooses the next node to visit in the graph based on a probabilistic function, leaving pheromone trails on the edges they visit. After each iteration of the algorithm, the pheromone values are updated based on the quality of the solution found by the ants, with the poor quality paths penalised and the high quality paths reinforced, leading the algorithm to eventually converge to optimality [15]. Zhang et al. [62] implemented the ACO algorithm with the purpose of planning a path for a UAV to follow, finding that the algorithm returned a path quickly and effectively, but more work needed to be done to smooth the path returned by ACO.

GAs are inspired by the process of natural selection in nature. A random set of candidates is generated, and explore the search space until they reach the destination point. Each of these candidates is then assessed based on a cost function or fitness function which is generally based on the length of the path. Some of the candidates are then selected for breeding, with the fitter candidates more likely to be selected, and combined with each other to produce new offspring candidates. A certain percentage of these new candidates are then mutated slightly before the next iteration begins to increase the diversity of the solutions returned. The algorithm terminates after a specified number of iterations or when a minimal threshold of diversity between the candidates is passed [26].

PSO algorithms are inspired by the behaviours of flocks of birds. Each particle is it's own agent in the search space but gets updated based on not only it's own experience, but also the experience of the best particle in the swarm, which is often evaluated by a fitness function, similar to the GA [44]. In path planning, each particle represents a path to the target and each particle moves or is updated by modifying a part of it's path. The algorithm is effective at quickly converging to the optimum path, but can also converge to only find a local optima [52].

Roberge et al. [48] implemented both a GA and a PSO algorithm for comparison in calculating flight paths. To implement these two functions a cost function is required, which is defined in the paper using multiple factors, including path length, average altitude, penalising paths through danger zones, penalising paths that consume more than the maximum power of the drone used and paths that consume more than the maximum fuel, paths that collide with the ground, and paths that can not be smoothed using circular arcs. The conclusion taken from this paper is that the GA as it is implemented by the authors returns optimal paths more frequently than the PSO algorithm.

2.3.3 Sampling-Based Algorithms

Sampling-based algorithms randomly sample the provided environment, and then use these samples to build a graph to represent possible paths between the start and target nodes [25]. The two most commonly used are Probabilistic Roadmaps (PRM) and Rapidly-exploring Random Tree (RRT).

PRM algorithms randomly generate a set of sample point and then connects these together based on specified constraints, such as connection distance and ensuring connections are collision-free [27]. PRM implementations are usually combined with a search algorithm, for example A*, to find the optimal path [60]. See illustration of PRM in *Figure 1*, taken from [38].

RRT algorithms take a random sample of points in the space, but then connects these points to the nearest part of the tree, without violating any constraints, which enables large parts of the space to be explored quickly. In path planning, two RRTs are grown simultaneously, one from the start point and one from the end point until they are connected, meaning a path has been found [29].

Yan et al. [60] implemented a PRM to build a map of the environment, and then used A* to calculate the optimal path through this map. The experimental results in the paper conclude that the method is able to return collision-free paths and is valid in complex three-dimensional environments.

Lin et al. [32] implement three variations of a closed-loop RRT algorithm for use in UAV path planning, finding the methods able to generate collision-free paths in real time when encountered with dynamic obstacles.

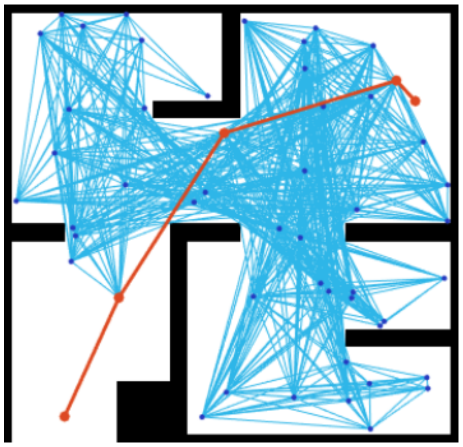


Figure 1: Example illustration of PRM Algorithm

2.3.4 Graph-Search Based Algorithms

Graph-search based algorithms are the more traditional approach to path planning, with numerous algorithms such as the basic breadth-first search and depth-first search algorithms [46], to more complex algorithms such as Dijkstra and A* [30] having been developed over the years. A* also has many variants which offer different benefits and disadvantages, such as Theta* [12], Lazy Theta* [41], Dynamic Anytime A* [31].

Danancier et al. [11] discuss the use of UAVs in a military context, attempting to find paths the UAV can follow that avoid threats such as radar detection systems in an air defence system, comparing Dijkstra's algorithm with a heuristic algorithm, the waypoint generation algorithm. The conclusion is that while the waypoint generation algorithm has much lower computational time, it is only comparable to Dijkstra when the number of threats in the environment is low.

Chiciudean et al. [7] investigates the path planning of a UAV in a continuous three dimensional space by converting the space into a disrete voxel space. Voxels are similar to pixels but with a third dimension. An A* algorithm is then applied to this discretized space, returning the optimal path in terms of distance covered. The path returned is then modified or smoothed to reduce unnecessary steering maneuvers for the UAV, which are returned as a result of the voxel space used. An example can be seen in *Figure 2*, where the red path is the path returned by A*, and the optimal path is shown in green. This smoothed path is found to reduce the path length returned by the A* method for navigation in a continuous environment.

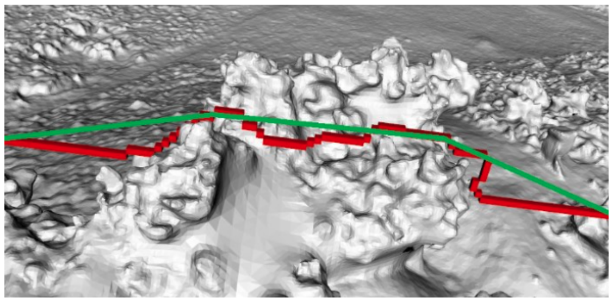


Figure 2: A* Path on Voxel Space

Similarly to Chiciudean et al. [7], Mandloi et al. [35] concludes that A* is not optimal with regards to a real continuous environment and compares two variants of A*, Theta* and Lazy Theta* using two metrics, computational time and the length of the generated path. The results indicate that the paths returned by Lazy Theta* are shorter than those returned by A* and the same as Theta*, while the computational time for Lazy Theta* is shorter than Theta* but longer than A*. The authors conclude that Lazy Theta* is the preferred algorithm.

2.4 Research Gap

Many of the papers mentioned above seem to only consider one or two factors when considering optimality, be it the path length, the computational time, or the number of turns in the path, the number of ascended and descended. This project looks to implement several algorithms, including the graph-search based algorithms Greedy and A*, to find the optimal path that will try to minimise all of the above mentioned factors.

2.5 Project Requirements

At the beginning of the project, these requirements were set out as some of the objectives to be achieved throughout the project. These will be discussed further during the report.

2.5.1 Core Functional Requirements

1. The program must calculate the most efficient route from any point A to any point B on a given topographic map, and output this path along with the estimated time taken.
2. The program must ensure this path will allow the UAV to travel without obstruction, e.g. the altitude of the UAV must be greater than the height of a hill or mountain at any point on the map.
3. The program must have an interface to input details, such as start and end points, drone information (battery, maximum height, strength, etc.), and topographic maps.

2.5.2 Non-Core Functional Requirements

4. The program should consider the battery level of the UAV when evaluating the flight plan.
5. The program should output:
   (a) an estimate of the time taken for the journey.
   (b) the distance covered by the drone.
6. The program could allow the user to define the preferred path, e.g. shortest path (distance covered), quickest path, most battery-efficient path.
7. The program could display the path on a front-end visualisation of the map.

3 Design

Section 3 describes the topographic maps and the algorithms designed as part of this study. Six algorithms are described (Section 3.2) including *Obstacle Avoidance using Circles* (Section 3.2.1), *Obstacle Avoidance using Waypoints within Circles* (Section 3.2.2), *Follow the Edge* (Section 3.2.3), *Greedy* (Section 3.2.4), *A** (Section 3.2.5), and *Bisect Obstacles* (Section 3.2.6).

3.1 Topographic Map

There exists a variety of publicly available topographic maps, including NASA's Shuttle Radar Topography Mission (SRTM) [39], Geological Survey Ireland (GSI) which provides very accurate and precise data [22], but the one used in this project is from the Japan Aerospace Exploration Agency (JAXA) [24]. The data for this map was acquired from millions of data images captured by the Advanced Land Observing Satellite "DAICHI" (ALOS). The primary reason for choosing this map was the size of the map files. The size of the GSI map for all of Ireland was several terabytes in size, while the ALOS map of Ireland comes to around one gigabyte.

3.2 Algorithm Designs

This section describes six algorithms, the first three of which are self-developed, while *Greedy* and *A** algorithms have been adapted to use in the context of the project, while the concept for the final algorithm, *Bisect Obstacles*, was inspired by an algorithm from Danancier et al. [11].

3.2.1 Obstacle Avoidance using Circles

The *Obstacle Avoidance using Circles* algorithm presents a very simple approach to the pathfinding problem for UAVs. The algorithm finds all obstacles on the path, draws a circle around each obstacle and then tries to follow one of the semi-circles until it reaches the other side of the obstacle. In the case both semi-circles still encounter an obstacle on their path, the radius of the circle increases until a suitable radius is found. (See *Algorithm 1*).

The advantage to this method is the quick computational time and the relatively simple implementation. It satisfies requirements 2, 5(a), 5(b), and 7 (Section 2.5).

Algorithm 1 Obstacle Avoidance using Circles Algorithm

```
1: $line \leftarrow getStraightLine(start, target)$
2: $max\_alt \leftarrow$ some arbitrary maximum altitude value
3: $obstacles \leftarrow getObstacles(max\_alt, line)$
4: $path \leftarrow [start]$
5: while $obstacles \neq \emptyset$ do
6:     $obstacle \leftarrow obstacles[0]$
7:     $centre\_point \leftarrow obstacle\_centre$
8:     draw $circle$ with arbitrary $radius$ around $centre\_point$
9:     split $circle$ into two $semi\_circles$
10:    check for a $route$ around either $semi\_circle$ that does not violate $max\_alt$
11:    if $route$ found then
12:        $path \leftarrow path + route$
13:        $obstacles \leftarrow obstacles - obstacle$
14:    else
15:        increase $circle$ $radius$
16:    end if
17: end while
```

However this method is not guaranteed to return a path, especially when the maximum altitude of a drone is set rather low. The radius of the circle could become larger than the distance from the start point to the destination point when faced with complex terrain such as a mountain range. Another issue with the path returned is that it requires the drone to be constantly making turns with small angles whilst avoiding obstacles. This can cause excessive and unnecessary battery consumption.

An example of the path returned by *Algorithm 1* can be seen in *Figure 3*.

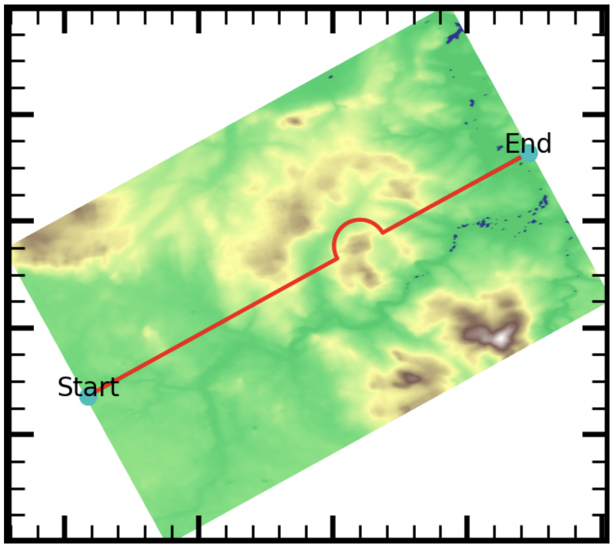


Figure 3: Obstacle Avoidance using Circles Path Returned - Example

The next section looks at a slight variant of the current algorithm, *Obstacle Avoidance using Waypoints within Circles*

3.2.2 Obstacle Avoidance using Waypoints within Circles

Algorithm 2 Obstacle Avoidance using Waypoints within Circles Algorithm

```
1: $line \leftarrow getStraightLine(start, target)$
2: $max\_alt \leftarrow$ some arbitrary maximum altitude value
3: $obstacles \leftarrow getObstacles(max\_alt, line)$
4: $path \leftarrow [start]$
5: while $obstacles \neq \emptyset$ do
6:     $obstacle \leftarrow obstacles[0]$
7:     $centre\_point \leftarrow obstacle\_centre$
8:     draw $circle$ with arbitrary $radius$ around $centre\_point$
9:     find $points$ at right angles to $centre\_point$ and $obstacle\_start$
10:    check for a $route$ through either $point$ that does not violate $max\_alt$
11:    if $route$ found then
12:        $path \leftarrow path + [obstacle\_start, point, obstacle\_end]$
13:        $obstacles \leftarrow obstacles - obstacle$
14:    else
15:        increase $circle\ radius$
16:    end if
17: end while
```

Obstacle Avoidance using Waypoints within Circles is similar to *Obstacle Avoidance using Circles* but differs through the use of waypoints within the circles. As in the previous algorithm, all obstacles on the path are found, with a circle then drawn around them. The difference is that for each circle, this algorithm (Algorithm 2) connects the starting point and exit point of each obstacle to two points on either side of the circle. (See Figure 4).

This method improves on the turning issue faced by the previous algorithm, as well as retaining the quick computational time. It is just as simple to implement, and satisfies the same requirements 2, 5(a), 5(b), and 7 (Section 2.5).

However, the problem that the algorithm can not compute a path for complex terrain has not been solved, nor does either algorithm have any guarantee of optimality.

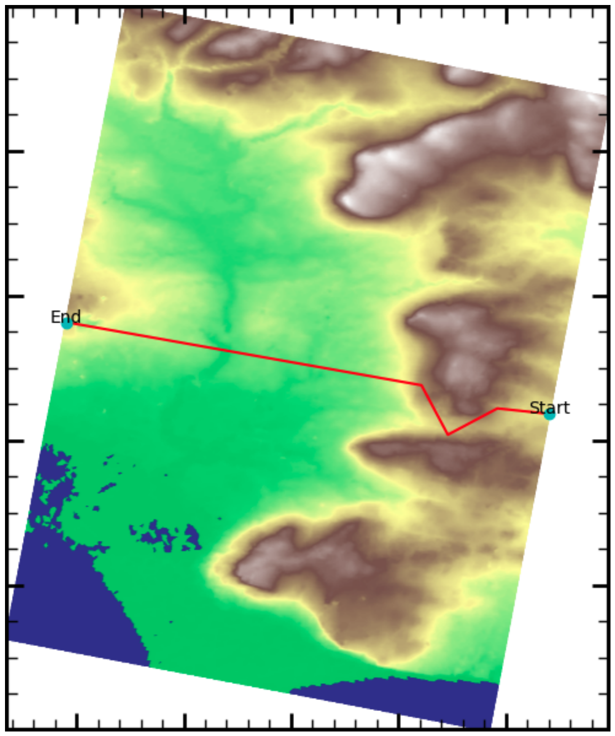


Figure 4: Obstacle Avoidance using Waypoints within Circles Path Returned - Example

3.2.3 Follow the Edge

Algorithm 3 Follow the Edge Algorithm

```
1: $line \leftarrow getStraightLine(start, target)$
2: $max\_alt \leftarrow$ some arbitrary maximum altitude value
3: $obstacles \leftarrow getObstacles(max\_alt, line)$
4: $path \leftarrow [start]$
5: for each $obstacle \in obstacles$ do
6:     find $route$ from $obstacle\_start$ to $obstacle\_end$ that does not violate $max\_alt$
7:     check that a $route$ through either $point$ doesn't violate $max\_alt$
8:     if $route$ found then
9:         $path \leftarrow path + [obstacle\_start, point, obstacle\_end]$
10:        $obstacles \leftarrow obstacles - obstacle$
11:    else
12:        increase $circle\ radius$
13:    end if
14: end for
```

Follow the Edge attempts to build on *Obstacle Avoidance using Circles* and *Obstacle Avoidance using Waypoints within Circles* by ensuring that if a path exists it can be found. Similarly to the previous algorithms (Algorithm 1, Algorithm 2), the *Follow the Edge* algorithm (Algorithm 3) first finds all obstacles that lie on the straight-line path, L, between the two inputted points. The algorithm then iterates through each of these obstacles, searching for a path around them while ensuring the drone does not violate its maximum altitude. When a path around each obstacle has been found, the algorithm returns the overall path.

Figure 5 illustrates an example of the type of terrain the *Follow the Edge* algorithm can traverse which the *Obstacle Avoidance using Circles* and *Obstacle Avoidance using Waypoints within Circles* algorithms can not. It is simple to see the path that *Follow the Edge will take*, while using a circles-based algorithm, the radius of either circle will not be restrained to a reasonable size to navigate around the obstacles. This is of course a rare example in practice but it clearly depicts the advantages to this algorithm. *Follow the Edge* satisfies requirements 2, 5(a), 5(b), and 7 (Section 2.5). See an example of a path returned by *Follow the Edge* in *Figure 6*.

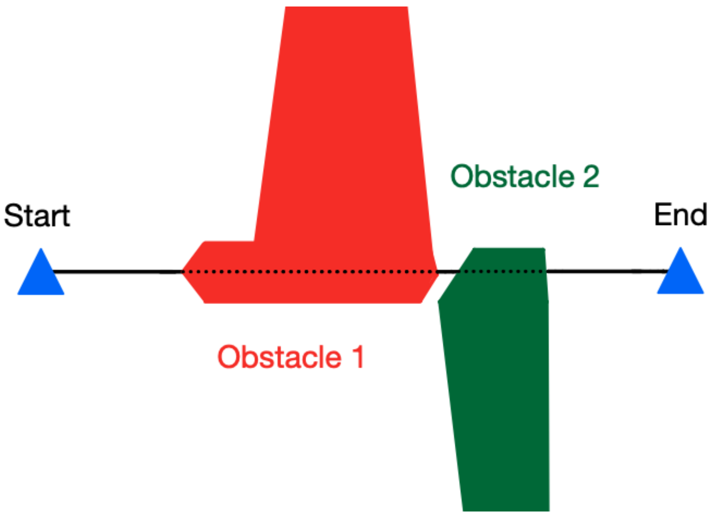


Figure 5: An example where the Follow the Edge algorithm is successful

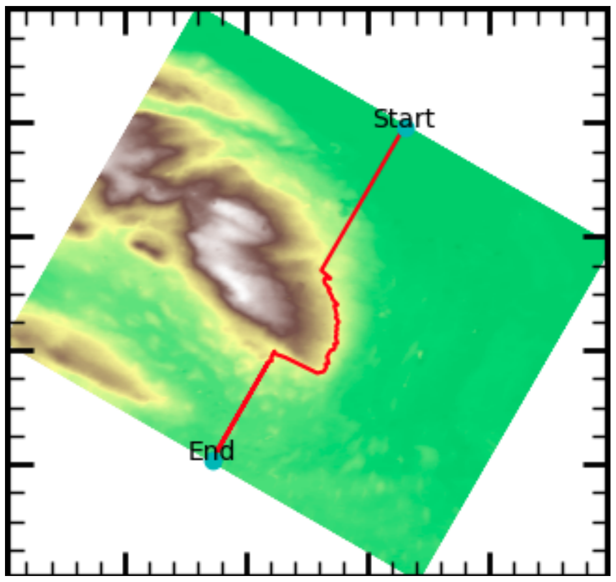


Figure 6: Example of Path Returned by Follow the Edge

However *Algorithm 3* has one fundamental flaw which causes it to fail when it encounters more complex terrain. Often, the part of the obstacle that is intersecting with L is only a small portion of the entire obstacle and the algorithm may have to search outside of the range of the obstacle start and obstacle end points. However, as it is described above, *Follow the Edge* is unable to do this, and hence fails to return a path which adheres to the maximum altitude of the UAV in these situations.

3.2.4 Greedy

Follow the Edge is conceptually quite similar to a *Greedy* algorithm. This leads to the next logical step being to implement a *Greedy* pathfinding algorithm. Greedy pathfinding algorithms take in a graph with vertices and edges, a start point and an end point. There are generally costs associated with either each edge or each vertex, which in this project is the Euclidean distance to the target. Each vertex stores an adjacency list which keeps track of the vertices that can be travelled to or the edges that can be traversed from this vertex. A greedy algorithm always chooses the vertex that has the lowest cost to reach the end. The graph given to the greedy algorithm is a grid representation of the terrain between the *start* point and the *end* point, with each point's adjacency list consisting of the four points surrounding it in the grid. Each point in the adjacency list is added to a priority queue, ordered by the cost function value, in other words ordered by the Euclidean distance to the target. The next point to move to is always the point with the lowest cost, the one at the front of the queue.

This algorithm satisfies requirements 2, 5(a), 5(b), and 7 (Section 2.5).

See *Figure 7* for an example of a path returned by *Greedy*.

This algorithm is the first guarantee completeness, meaning that if a path exists, *Greedy* will find it. However it can not guarantee optimality.

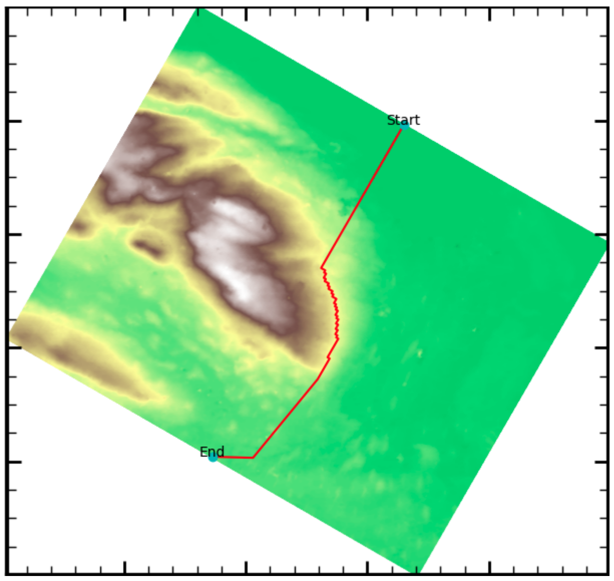


Figure 7: Example of Path Returned by Greedy

3.2.5 A*

For optimality, the *A** algorithm is implemented. A typical A* algorithm uses a combination of the cost of the path so far and a heuristic to estimate the distance to the end, for example the Euclidean distance. This heuristic must be admissible to ensure optimality, meaning that it does not overestimate the distance. Combining these two gives the cost function that A* typically uses. However the objective of this project is not only to find the path with the shortest distance, but also to minimise the number of changes in directions and the number of metres ascended and descended. Hence the *A** algorithm used in this project is enhanced to include angle and altitude information. The cost function used by *A** is as follows, with the current point being p_0 and the neighbour point being p_1:

$$potential\_cost_{p_1} = cost\_to\_here + dist\_to\_target + energy\_estimate \tag{1}$$

where $cost\_to\_here$ is calculated as:

$$cost\_to\_here = cost_{p_0} + 0.1 + (energy\_used \cdot (1 + factor)) \tag{2}$$

where $factor$ is the value calculated from the drone flying above maximum altitude and the 0.1 value is 100 metres converted to kilometres. Using alt_diff as the difference in metres between the specified flying altitude of the UAV and the altitude of p_1, and max_alt_diff as the difference in metres between the flying altitude and the maximum altitude of the drone, $factor$ can be calculated as follows:

$$\begin{aligned} factor &= alt\_diff^2 \cdot \frac{1000}{max\_alt\_diff^2} \\ factor &= \frac{factor}{100} \end{aligned} \tag{3}$$

$energy\_used$ is calculated by getting the energy consumed for 100 metres (further detailed in 5.1.3), and then multiplying this by 20% if the drone has made a turn.

The $energy\_estimate$ value in Equation 1 is calculated by multiplying $dist\_to\_end$ by the epm value, as described in Section 5.1.3.

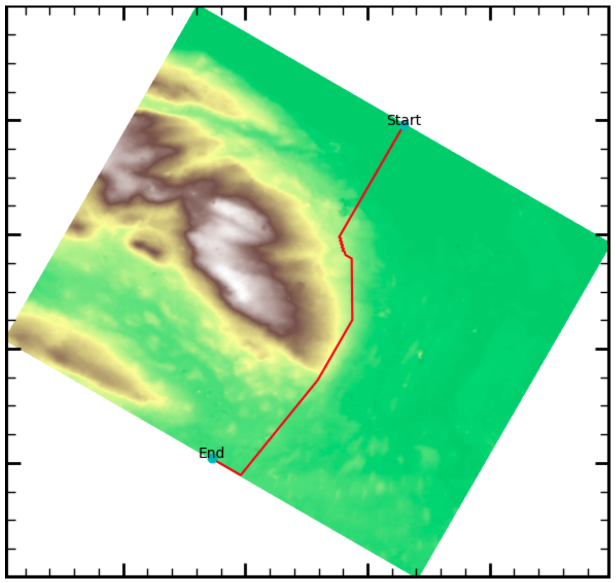


Figure 8: Example of Path Returned by A*

The modified *A** algorithm can guarantee completeness and optimality with respect to distance covered, number of turns, and number of metres ascended and descended.

This algorithm satisfies requirements 1, 2, 4, 5(a), 5(b), and 7 (Section 2.5).

See *Figure 8* for an example of a path returned by A*.

The only issue with this algorithm is the computational time. Due to the increased number of factors to consider, this algorithm takes much longer than all previous algorithms to compute. This could be an issue if the UAV is to encounter an unexpected obstacle on its path, and must recalculate a path around it.

3.2.6 Bisect Obstacles

Algorithm 4 Bisect Obstacles Algorithm

```
1: if $target \in waypoints$ then
2:     $waypoints$
3: end if
4: $line \leftarrow getStraightLine(start, target)$
5: $max\_alt \leftarrow$ some arbitrary maximum altitude value
6: $obstacles \leftarrow getObstacles(max\_alt, line)$
7: if $obstacles \neq \emptyset$ then
8:     $obstacle \leftarrow obstacles[0]$
9:     $halfway\_point \leftarrow obstacle\_centre$
10:    draw $perpendicular\_line$ from $halfway\_point$
11:    find first $points$ on either side of $perpendicular\_line$ that are below $max\_alt$
12:    connect $points$ to $start$ to create $new\_lines$
13:    if either $new\_line$ has no obstacles then
14:        connect $point$ to $target$ to create $new\_line$
15:        if $new\_line$ has no obstacles then
16:            $path \leftarrow path + point$
17:        else
18:            $Bisect(point, target, waypoints)$
19:        end if
20:    else
21:        $Bisect(point, target, waypoints)$
22:    end if
23: end if
```

The *Bisect Obstacles* algorithm (4) was implemented to combat the computational time issues of *A**, without the completeness issues of either of the *Circles* algorithms and *Follow the Edge*. It was inspired from a similar algorithm designed in Danancier et al. [11], who use it in a military application, where the obstacles are modeled as circles to avoid. To be implemented in this project, the algorithm had to be adjusted to navigate around more complex shapes.

Bisect Obstacles fulfills requirements 2, 5(a), 5(b), and 7 (Section 2.5).

While it can not guarantee the path is optimal, as it does not take altitude or turns into consideration, it should return a near-optimal path in terms of distance covered in much shorter time than $A*$.

4 Implementation

This section describes the programming language used (Section 4.1), the challenges faced when accessing a topographic map (Section 4.2) and the data within it (Section 4.3), the main data structures (Section 4.4), and implementing all algorithms (Section 4.5).

4.1 Programming Language Used

All of the algorithms in this project were implemented using Python. Python was chosen because of the wide range of libraries and packages available, which made accessing the data in the topographic map rather simple, specifically *gdal* [58], and *haversine* [51].

4.2 Accessing a Topographic Map

Geological Survey Ireland provides a very precise map of Ireland (publicly available on [22]). However, this map consists of thousands of files and each one must be downloaded separately. The solution to this problem is to build a web scraping program, which was implemented in Python. Once built, the program was executed and around 600 files were initially downloaded, approximately 140 gigabytes in size.

Due to the file size, it was decided to search for more maps such as NASA's SRTM [39] and JAXA's ALOS mission [24]. While these were less detailed, they were still sufficiently accurate for use in this project with a spatial accuracy of around 30 metres [24].

The topographic data used in this project was supplied by JAXA's ALOS.

4.3 Accessing the Data in the Map

The map of Ireland provided by JAXA is split into into 25 tiles. Each of these tiles is stored in three different files, the file name ending with either DSM, STK, or MSK. The only file used in this project ends in DSM. The other two are supplementary data, with the STK files providing information about the number of images that were used to compose the data, and the MSK files indicating whether the corresponding value in the DSM file is valid or invalid.

To access the data stored in these *.tif* files, the Python package called *gdal* [58] must be used. Using this package, an input geographical coordinate can be converted to a pixel coordinate which can be used to access a single pixel in the file, storing the desired altitude information.

There are 25 different files where a coordinate could be stored, so a method to locate the relevant file storing the coordinates must be found. Upon performing the conversion with *gdal*, the pixel coordinates returned will be in the form *(x, y)*, where both x and y have the range *(0, width/height of file)*. Each of the files used in this project have the same width and height of 3600. Hence, if x or y is less than zero or greater than 3600, the coordinates can not be contained in this file.

This information can also be used to estimate which file to try next. For example, if the y value is greater than 3600, than the coordinate is likely located in the file above the current one. The files are named as follows:

$$ALPSMLC30\_N051W008\_DSM.tif$$

where the N051 component of the name is the top-left latitudinal coordinate and the W008 component is the corresponding longitudinal coordinate. As in the example above, if the y value is greater than 3600, then the coordinated is located in a file north of the current file, which means the longitudinal value in the file will increase by one. Hence, the next file to check is $ALPSMLC30\_N052W008\_DSM.tif$ with a longitudinal value of 52°.

It is also possible that the input geographical coordinates will be outside the range of the topographic map, in which case an error will be raised.

4.4 Data Structures and Classes

Throughout the project, several classes were implemented to be used in different algorithms, including the Line class (Section 4.4.1), the Obstacles class (4.4.2), the Circles class (Section 4.4.3), and the Area Map class (Section 4.4.4).

4.4.1 Line Class

The Line class is used in every algorithm and was the first class implemented. It receives two points, a start and an end point, a *step_dist* parameter and retrieves a dictionary with coordinates as keys and the altitude of the coordinates as values. Each set of coordinates is separated by *step_dist* metres, typically 100 metres.

To retrieve this dictionary, the class needs to know each coordinate between the start point and the end point. To do this, the Line class calculates the distance in kilometres from start to end using the haversine formula [51], which accounts for the curvature of the Earth during calculations. The next step is to divide this distance by the *step_dist* value converted to kilometres to get the total number of steps that must be made to reach the target, *total_steps*. *Algorithm 5* illustrates the method.

Algorithm 5 Get Line

```
1: $dist \leftarrow haversine(start, end)$
2: $total\_steps \leftarrow dist/step\_dist$
3: $lat\_diff \leftarrow$ (latitude of $start$) - (latitude of $end$)
4: $lon\_diff \leftarrow$ (longitude of $start$) - (longitude of $end$)
5: $lat\_increment \leftarrow lat\_diff/total\_steps$
6: $lon\_increment \leftarrow lon\_diff/total\_steps$
7: $lat \leftarrow$ (latitude of $start$)
8: $lon \leftarrow$ (longitude of $start$)
9: $line \leftarrow \{\}$
10: $lat\_end \leftarrow$ (latitude of $end$)
11: while $lat < lat\_end$ do
12:     $line \leftarrow line + \{(lat, lon) : altitude(lat, lon)\}$
13:     $lat \leftarrow lat + lat\_increment$
14:     $lon \leftarrow lon + lon\_increment$
15: end while
```

4.4.2 Obstacles Class

This class takes in a Line object, a maximum altitude value and returns a dictionary. The line is analysed for any points above the maximum altitude. If there is a series of points in a row, only the first point, halfway point and last point of this series of points will be stored. See Algorithm 6.

Algorithm 6 Get Obstacles

```
1: $line \leftarrow$ get $Line$ from $start$ to $end$
2: $too\_high \leftarrow$
3: $obstacles \leftarrow$
4: for $point, altitude in line$ do
5:     if $altitude >= max\_altitude$ then
6:         $too\_high \leftarrow point + too\_high$
7:     else if $too\_high.length > 0$ then
8:         $halfway\_index \leftarrow too\_high.length/2$
9:         $obstacles \leftarrow obstacles + [too\_high[0], too\_high[halfway\_index], too\_high[-1]]$
10:        $too\_high \leftarrow$
11:    end if
12: end for
```

4.4.3 Circles Class

The next class to implement is the Circles class which is used in both the *Obstacle Avoidance using Circles* and *Obstacle Avoidance using Waypoints within Circles* algorithms. Taking two inputs, *centre* and *radius*, the class returns a dictionary containing all points in the circle, with the value being the altitude at that point.

To get the x and y coordinates of any point on a circle given the radius, *r*, *centre*, and angle from the centre in radians, θ, the following formula can be used:

$$\begin{aligned} x &= centre\_x + (r * cos(\theta)) \\ y &= centre\_y + (r * sin(\theta)) \end{aligned} \tag{4}$$

Using this equation, it is simple to draw the entire circle, using a loop to increase the angle until the angle reaches 180°, updating the dictionary with two points each time. For each θ value, two points on a circle can be calculated, the point at θ and the point at $360 - \theta$.

4.4.4 Area Map Class

The *Greedy* and *A** algorithms require a graph to find a path to the target. The Area Map class will provide such a graph by creating a grid of the area between the start point and the end point. The input is a *start* point, an *end* point, and a *width* value.

The first step is to create a Line object using the start and end points, called line A. To get the rest of the grid, *width* points to the left and right of line A must be calculated. To find a point perpendicular to the line, the *lat_increment* and *lon_increment* values from the Line class can be used. If the slope of line A is m, then the slope of a perpendicular line B is $-\frac{1}{m}$. In this project, the slope of a line can be expressed as $\frac{lat\_increment}{lon\_increment}$. Hence to get a point p perpendicular to line A, use slope:

$$-\frac{lon\_increment}{lat\_increment} \tag{5}$$

or:

$$\begin{aligned} lat_p &= lat_{start} - (lon\_increment_A * \frac{width}{2}) \\ lon_p &= lon_{start} + (lat\_increment_A * \frac{width}{2}) \end{aligned} \tag{6}$$

The plus and minus in (4) can be interchanged. This simply changes the direction of the line, i.e. as it is in (4), the point will be to the "left" of the start point, while interchanging the signs will retrieve a point to the "right". This equation retrieves a point that is $(\frac{width}{2})$ steps perpendicular to the *start* point.

In order to retrieve a line that is the required width, repeat the formula but switching the plus and minus signs to get a point on the other side of *start*. Then instantiate a Line instance between these two points, storing the output line in a list called *grid*.

This process can then be repeated for each element of line A, until all points in the grid have been found and the grid can be returned.

4.5 Algorithm Implementations and Challenges

With knowledge of the above classes and data structures, any challenges encountered during the implementation of the algorithms can be more easily understood. In this section, the implementation of each algorithm is described, including both of the algorithms using circles (Section 4.5.1), the *Follow the Edge* algorithm (Section 4.5.2), the *Greedy* and *A** implementations (Section 4.5.3), and the *Bisect Obstacles* implementation (Section 4.5.4).

4.5.1 Algorithms using Circles

The first two algorithms implemented were the *Obstacle Avoidance using Circles* algorithm and the *Obstacle Avoidance using Waypoints within Circles* algorithm. The first challenge to overcome is to do with splitting each circle into two semi-circles, which is necessary for both algorithms. The challenge here is that each circle is stored as a list of points, so it is unclear which points should be in one semi-circle and which should be in the other. The first attempt at solving this problem involved finding the two semi-circles when building the circle. To do this, calculate the angle of the line between the start and end points with the line of latitude being 0°. This angle can then be used as the angle parameter, a, when building each circle in the path.

This solution can be seen in *Figure 9*, with one semi-circle containing the green points, and the other containing red points.

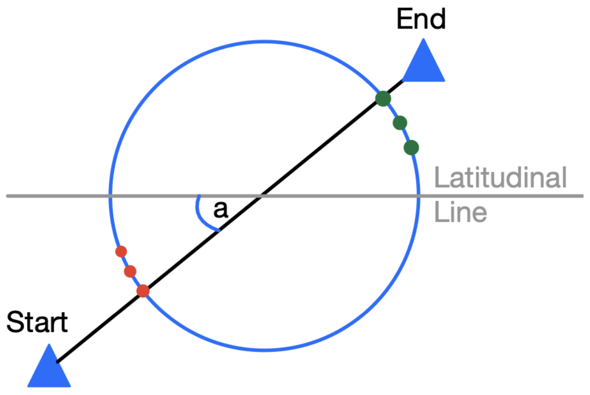


Figure 9: Illustration of Semi-Circle - Solution 1

It became clear when implementing this solution that it was unnecessarily complex.

The more simple and effective solution is to draw the circle with angle 0° and then store all points on the circle in the same list. To split the circle in half, the point that is closest to the *start* point must be found. This can be done using a simple for loop. Once this point has been found, both semicircles can be retrieved by slicing the list in half.

The next unforeseen challenge was caused by a flaw in the algorithm itself. It is possible that, when dealing with complex terrain, one circle C_0 can become much larger than another C_1, so much so that C_1 will be contained within C_0. When this happens, the path will lead the drone around C_0 until it reaches the end, where it will then go back away from the target to reach the beginning of C_1. Therefore, the drone will have to fly over the obstacle that C_1 is trying to avoid.

This is a more difficult problem to solve, as the issue lies within the algorithms themselves. Removing any contained points is the obvious solution, but this can cause further complications. *Figure 10* illustrates the case where removing the circle to avoid obstacle 2 is problematic.

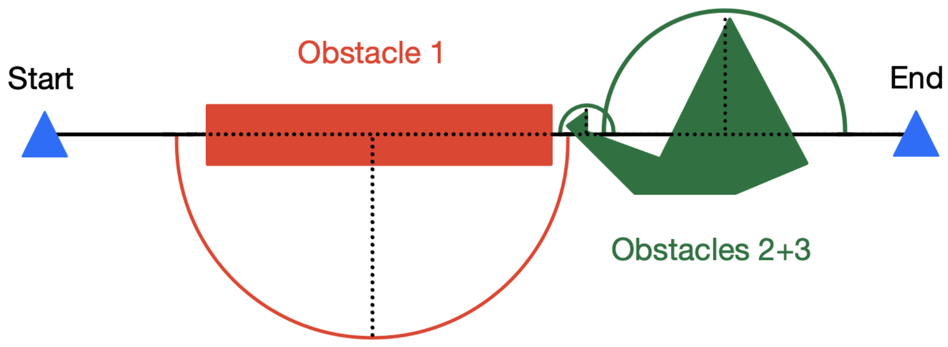


Figure 10: Circles Algorithm Problem Illustration

The implementation of *Obstacle Avoidance using Waypoints within Circles* has got a simple method (Algorithm 7) to remove contained circles, receiving the list *waypoints* which contains the start point, end point and all semi-circles on the path, while the *Obstacle Avoidance using Circles* algorithm keeps all contained circles, to be noted for Section 5.

Algorithm 7 Remove Contained Circles

```
 1: $i \leftarrow 1$
 2: $circle\_length \leftarrow 3$   ▷ 3 because this is the Waypoints algorithm, so only 3 waypoints per semi-circle
 3: while $i < waypoints.length$ do
 4:     $circle1 \leftarrow waypoints[i : i + circle\_length]$
 5:     $distance\_to\_start1 \leftarrow haversine(start, circle1[0])$
 6:     $distance\_to\_end1 \leftarrow haversine(end, circle1[-1])$
 7:     $j = 1$
 8:     while $j < waypoints.length$ do
 9:         $circle2 \leftarrow waypoints[j : j + circle\_length]$
10:         $distance\_to\_start2 \leftarrow haversine(start, circle2[0])$
11:         $distance\_to\_end2 \leftarrow haversine(end, circle2[-1])$
12:         if $distance\_to\_start1 > distance\_to\_start2$ & $i < j$ then   ▷ then remove circle1 from waypoints
13:             $waypoints \leftarrow waypoints[: i] + waypoints[i + circle\_length :]$
14:             $i \leftarrow i - circle\_length$
15:         else if $distance\_to\_end1 < distance\_to\_end2$ & $i < j$ then   ▷ then remove circle2 from waypoints
16:             $waypoints \leftarrow waypoints[: j] + waypoints[j + circle\_length :]$
17:         end if
18:         $j \leftarrow j + circle\_length$
19:     end while
20:     $i \leftarrow i + circle\_length$
21: end while
```

The next algorithm tries to solve the issues of completeness with the two circles algorithms.

4.5.2 Follow the Edge Algorithm

Following on from the previous algorithms, the key idea for this algorithm is to find all obstacles in the path, and then find a way around them by staying at the same altitude. The same method for finding all obstacles can be used, but finding a way around them must be designed.

First, an empty list to store waypoints, *waypoints*, must be created.Then for each obstacle on the path, create a Line instance, L_O, between the start of the obstacle and the end, where this line has a smaller value for the *step_dist* parameter than the line from *start* to *end*, L, for example 50 metres. Store the start of the obstacle in *waypoints*. For each point in L_O, move perpendicularly to L until a point, p_0, is found that doesn't exceed the maximum altitude and store this point in *waypoints*.

This algorithm is illustrated in *Figure 11*.

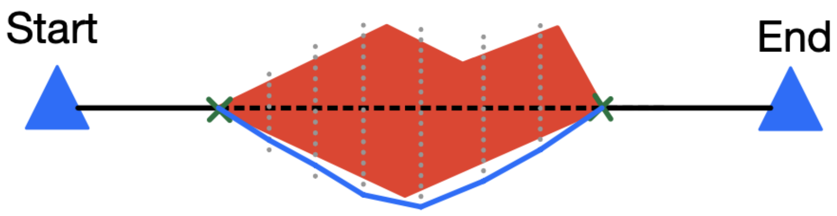


Figure 11: Follow the Edge Illustration

During the testing of this algorithm, it was discovered that the paths returned do not always adhere to the maximum altitude of the drone. To try and solve the problem, when a point p_0 is found, a Line is drawn between p_0 and the previous point in *waypoints* with an even smaller *step_dist* value, ten metres, to ensure any obstacles in between can be found. If the maximum altitude of this line is acceptable, then p_0 can be added to *waypoints*. Otherwise if the line violates the maximum altitude, then find the next perpendicular point and try again.

After implementing the above step, the paths returned violated the maximum altitude less frequently but the problem persisted. To understand the problem better, some paths were imported into this website `https://www.google.com/maps/d/u/0/`, which takes an xml file of coordinates and plots them on Google Maps. Any mountains or mountain ranges near to the points on the path could now be seen, making the issue with the algorithm much clearer. The problem is illustrated in *Figure 12* and explained below.

The *Follow the Edge* algorithm searches the line L from the *start* point to the *end* point for any obstacles and stores the obstacle start and end points. The issue with this is that the start and end point of the obstacles will generally not be the actual start and end point of the obstacle, with respect

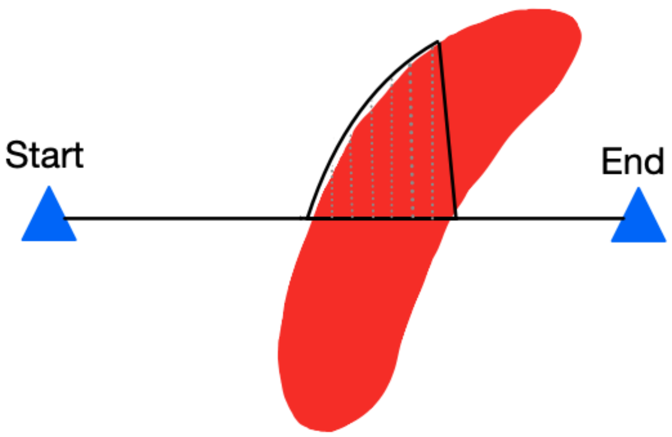


Figure 12: Problem with Follow the Edge

to L. The actual start and end point of obstacles may lie to the side of L. Therefore when *Follow the Edge* tries to find a path around these obstacles, it may only be searching for a path around a fraction of the obstacle.

A method to solve this problem with respect to the end of the obstacle was implemented, but not to solve the obstacle start problem. Once the algorithm has returned a path around an obstacle, a line, L_e between the obstacle end and the final point in *waypoints* is generated. If the maximum altitude of L_e is too high, then the end point of the obstacle will be moved closer to the *end* point of L and the same steps to find p_0 will be carried out. This can be repeated until either the obstacle end has surpassed the actual *end* point or until a path has been found to the obstacle end.

This solution did improve the results but the issue with the start of the obstacle remains. The algorithm is already needlessly complex as the key idea behind this algorithm is very similar to another well-known and simpler path-finding algorithm, *Greedy*. Therefore, instead of implementing the method for finding the start of the obstacle, the algorithm will be run first using the maximum altitude of the drone. If a path can not be found with this altitude, then the altitude is decreased by 40 metres. This leads to more obstacles being found on the path, increasing the chance of finding a collision-free path to the *end* point.

4.5.3 Greedy and A* Algorithms

Greedy and *A** are some of the most well-known algorithms used for path-finding in a two-dimensional space. All of the algorithms implemented thus far have mapped out the terrain in a two-dimensional space, so we can continue using the existing classes and methods for *Greedy* and *A**.

The first of the two algorithms implemented was *Greedy*. One of the major decisions to be made was how will the space be represented as a graph. The class Area Map was implemented to represent the area between the *start* and *end* points in a matrix.

Given a point p_0, the next decision to make is which points to access as p_0's neighbours. Given the grid format that the Area Map class returns, the index of p_0 within the matrix can be used to retrieve all neighbours. Each point will have four neighbours, the point closer to the *end* point, the point closer to the *start* point and two points to the side of p_0.

The algorithm iterates through each of the four neighbours, checking if the altitude is above the maximum altitude. If it is assessed to be an obstacle, the point is skipped and the next neighbour is checked.

The cost of each point in the matrix is the distance in kilometres to the *end* point. The *Greedy* algorithm always chooses the point with the lowest cost, i.e. the closest neighbour to the *end* point that is not an obstacle.

A dictionary, *path*, is used to store the path, where each key is a point with the value being the next point in the path.

The *Greedy* algorithm is the first implemented that guarantees completeness but it still can not guarantee optimality, unlike *A**.

Implementing *A** comes with more decisions to be made. *A** chooses the next point based on the cost of the path so far plus the predicted cost to the end. The distance between one point and any of its neighbours is always 100 metres, so the cost from the *start* point to each neighbour is always the cost to the current point, p_0, plus 100 metres or 0.1 kilometres.

Implementing A* also allows other factors to be considered, such as minimising the energy consumed. As stated in Section 5.1.3, the energy consumption is calculated using altitude flown and number of turns made. This is very useful as it can be added to the cost of the path to the current point, and also can be used to predict the cost to the *end* point.

The only issue with the *A** algorithm is that due to the increased number of factors, the computational time becomes much longer. The graph is much

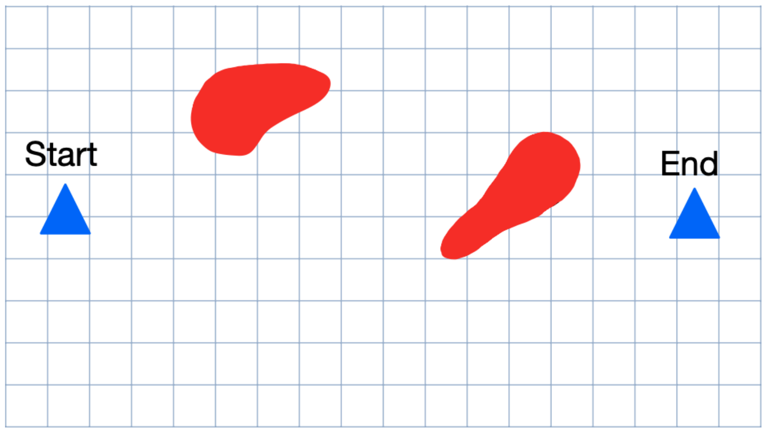


Figure 13: Example of A* Graph

bigger than it needs to be, because most of the space is covered by safe areas for the drone to fly in. For most paths, only a small percentage of the area, or nodes, will be taken up by obstacles, also contributing to the increase in time.

In the worst case, every point in the graph must be explored, with time complexity of O(n), where n is the number of points in the graph. The actual time complexity is, in practice, much lower than the worst case.

4.5.4 Bisect Obstacles

This algorithm was implemented to counter the time issues with *A**. It is a recursive algorithm which aims to bisect obstacles until a path to the *end* point has been found. An example of the algorithm can be seen in *Figure 14*.

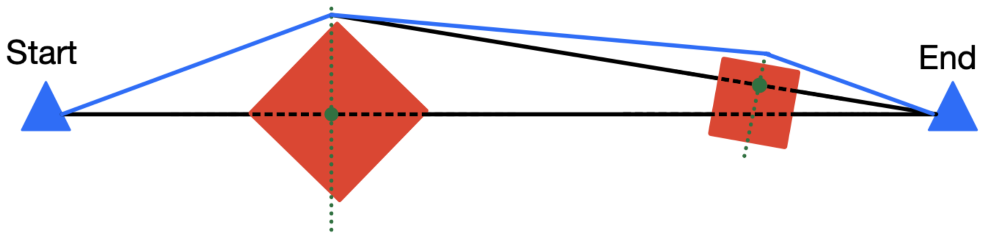


Figure 14: Bisect Obstacles Illustration

The first challenge to overcome is deciding on a heuristic to choose from the two points the bisect line returns. The heuristic implemented is to choose the point that is closest to the halfway point of the obstacle. This heuristic can be seen implemented in the second obstacle in *Figure 14*.

The challenge that could not be overcome due to time restraints is how to store the path from *start* to *end*. There could be numerous promising points explored that end up not contributing to the final path. The current implementation uses a dictionary similar to the *path* dictionary in the Greedy and A* implementations.

Another problem with the algorithm is the possibility of entering an infinite loop, by infinitely trying to navigate around the same obstacle. This problem is lessened with the heuristic used but is not eliminated.

The current implementation of the algorithm seems to work on simple obstacles, for example if the second obstacle in *Figure 14* did not exist and the algorithm only had to traverse one simple obstacle. However, when more obstacles are introduced, the method returns the *path* dictionary storing many unused points, which leads to a path unable to be found among these points.

4.6 Additional Methods Implemented

4.6.1 Generating an Obstacle Map as a Graph for A*

Reducing the size of the input graph in *A** (Section 4.5.3) could result in a massive gain in time. A different way to represent the area is to only store points that are above the flying altitude of the drone but below the maximum altitude allowed. This would massively reduce the number of nodes in the graph. However, the issue with implementing this is that each node requires an adjacency list.

This method takes in the Area Map generated above, and adds all points that are greater than or equal to the flying altitude but less than the maximum altitude to a dictionary, *obstacle_map*, where the key is the point and the value is the adjacency list. To retrieve the adjacency lists:

Algorithm 8 Adjacency List Retrieval

```
1:  for each $point1 \in obstacle\_map$ do
2:      for each $point2 \in obstacle\_map$ do
3:          if $point2 \neq point1$ then
4:              $line \leftarrow$ get $Line$ from $point1$ to $point2$
5:              if maximum altitude of $line < max\_alt$ then
6:                  $obstacle\_map[point1] \leftarrow obstacle\_map[point1] + point2$
7:                  $obstacle\_map[point2] \leftarrow obstacle\_map[point2] + point1$
8:              end if
9:          end if
10:     end for
11: end for
```

Assuming that getting a line between two points costs constant time, the complexity of this method is O(m^2), where m is the number of points in *obstacle_map*. In cases where the space between the *start* and *end* points contains very few obstacles, this method may prove more efficient than the grid-based implementation of *A**. However in most cases, this method drastically increases the overall computational time of *A** and is hence unusable.

4.6.2 Path Smoothing

This function was adjusted for use in this project from Danancier et al. [11]. It takes in a path, assessing it for any unnecessary points. *Algorithm 9* shows the algorithm as it appears in Danancier et al.'s paper [11].

Algorithm 9 Path Smoothing Algorithm

```
1: for each triple $(o, m, d)$ in $Path$ do
2:     if $m$ is a waypoint & $o$ and $d$ are not a base of UAV & there is no threat then
3:         Remove $m$ from $Path$
4:     end if
5: end for
6: return $Path$
```

In [11], they are searching for flight paths in a military context, hence the use of the words *threat* and *base*. The algorithm was adjusted slightly for use in this project, with the *threat* being replaced by the flying altitude of the drone. In other words if the middle point of the three points is below the flying altitude then it can safely be removed. (See Figure 15).

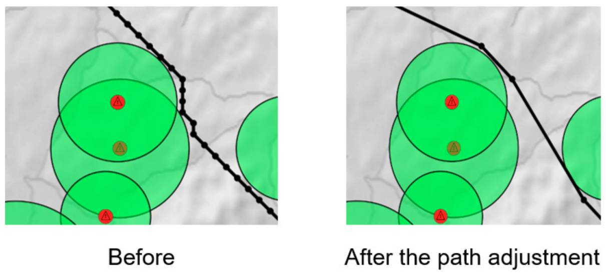


Figure 15: Path Smoothing Example from [11]

5 Evaluation

Section 5 compares all algorithms using graphical representations of the results, reporting and explaining the conclusions gathered from these results. The objective of these algorithms is to find the optimal path between two paths, considering a variety of factors, including distance covered, number of changes in direction, number of metres ascended and descended and overall energy consumption.

5.1 Outline of Tests Performed

The maximum altitude of the drone is always set to be the altitude of the *start* point plus 120 metres, so it will be different for each pair of points. This threshold is chosen so that the randomly generated points also present a wide variety of terrain types to be traversed, with the possibility of the maximum altitude being set at 700 metres, making the terrain simple to traverse, or set at 140 metres, introducing many obstacles to navigate.

Given the *Bisect Obstacles* algorithm has not been fully implemented, it will not be included in the tests.

Each algorithm will be compared using a set of 25 randomly generated pairs of points, P. Each of these pairs is less than 30 kilometres apart, $dist_{apart}$, and guaranteed to have an obstacle in between them. The average distance between the points is 21.12 kilometres. The paths returned were then analysed and several statistics were stored for each path, including number of turns, the average angle of each turn, the number of metres ascended and descended, the distance ratio, time taken in minutes, and the energy consumption in milliampere hours (mAh). The average of these numbers was then calculated over the 25 runs.

Each algorithm will be tested twice using the 25 points, the first time, the paths returned will not have any path smoothing applied to them (Section 4.6.2), while in the second run, each of the five algorithms tested will use path smoothing. Therefore, the differences between not only the five algorithms can be seen but also the effect of path smoothing on these algorithms.

5.1.1 Metrics Used - Completeness

A key metric to determine the completeness of each algorithm is if the maximum altitude set by the program was violated or not. If it is violated on algorithm A but not on algorithm B, then algorithm A can not be considered complete.

5.1.2 Metrics Used - Distance Covered

The distance covered on a path returned by the algorithm, $dist_{total}$ from start to finish must also be considered. Each algorithm is compared with the ratio:

$$\frac{dist_{total}}{dist_{apart}} \tag{7}$$

5.1.3 Metrics Used - Energy Consumption

Other minor metrics, such as average angle turned in degrees, number of turns, number of metres ascended and descended, as well as the distance travelled, can all be consolidated into one value, the energy consumption. There are numerous research papers attempting to find a single equation that can calculate the energy consumption for a drone, but none have yet found an equation [1, 56, 61] that is accurate for all drones and for all journeys. Zhang et al. [63] compares 11 of these equations, citing energy used per metre in joules, *epm*, for each equation. The figure used, 108.1 joules, for *epm* in this project was taken by averaging the 11 values provided in Zhang et al. [63]. Each metre ascended, $metre_{asc}$, costs an extra 30%, each metre descended, $metre_{desc}$, costs an extra 15%. The cost of a turn with angle θ is calculated as such:

$$epm \cdot \frac{\theta^{0.75}}{100} \tag{8}$$

To calculate the total energy consumed on a journey in joules:

$$energy = (epm \cdot dist_{total}) + (epm \cdot factor) \tag{9}$$

where $factor$ is calculated as follows:

$$(total\_metres_{asc} \cdot (epm \cdot 0.3)) + (total\_metres_{desc} \cdot (epm \cdot 0.15)) + \sum_{i=1}^{n} \theta_i^{0.75} \quad (10)$$

n = number of turns made;

5.1.4 Unit Conversion from Joules to Milliamp Hours

The unit mAh is used as it is generally the unit used for the energy capacity of a battery. While the energy consumption model used in this project (Equation 9) is used mostly for comparison purposes and not to estimate realistic energy consumption values, each path can still be assessed against the average battery capacity of a drone using the mAh value. To convert from joules (J) to mAh, joules must first be converted to watt hours (Wh):

$$Wh = joules/3600 \quad (11)$$

The conversion from Wh to mAh requires the voltage of the drone battery. The value used is 22.8 volts (V), taken from the voltage of the battery used in the DJI Matrice 100 [14]. With this value the mAh value can be calculated using the formula:

$$mAh = \frac{Wh}{voltage} * 100000 \quad (12)$$

5.2 Optimal Flight Path Analysis

Table 1 presents the findings from the data, where each algorithm was tested on the same 25 pairs of points. The paths returned were left as they were, there was no path smoothing applied to them (applied in Table 2). To improve the readability of the table, *Obstacle Avoidance using Circles* will be referred to as Circles and *Obstacle Avoidance using Waypoints within Circles* as Circles w/ Waypoints.

If the maximum altitude is violated on any path p, the energy consumed for p is doubled. This penalises algorithms that violate the maximum altitude more frequently.

Name	**#Times Max. Alt Violated**	**Comp. Time (s)**	**Distance Ratio**	**Energy Consumed (mAh)**
Circles	16	0.69	2.54	18412
Circles w/ Waypoints	13	0.18	1.35	7408
Follow the Edge	15	0.14	1.53	9720
Greedy	2	3.33	1.57	6536
A*	2	39.37	1.42	5042

Table 1: Results without Path Smoothing Applied

From Table 1, it is clear that the paths that A* returns consume much less energy than any of the other algorithms, with the distance travelled also lower on average than most of the other algorithms (1.42). The *Obstacle Avoidance using Waypoints within Circles* algorithm surprisingly has a lower distance ratio value (1.35) than *A**. However there is possibly a reasonable explanation for this. Both *Obstacle Avoidance using Circles* and *Obstacle Avoidance using Waypoints within Circles* often experience the issue where the circle size becomes too big and it expands beyond the boundary of the topographic map. When a point on the circle which is outside of the topographic map is accessed, the program raises an error. When this happens, the straight line path between *start* and *end* is given to both algorithms. The straight line path of course has a ratio of 1 with itself, hence decreasing the average value. Despite this certainly being a valid candidate for the cause of the low distance ratio, this error seemingly did not positively affect the distance ratio of *Obstacle Avoidance using Circles*. Regardless, the *Obstacle Avoidance using Waypoints within Circles* algorithms is certainly the most promising of the three algorithms designed in this project.

The issues with *A**'s increased computational time (39.37 seconds) also become clear when compared with the other algorithms, where *Greedy* has the next highest time (3.33 seconds).

Name	**#Times Max. Alt Violated**	**Comp. Time (s)**	**Distance Ratio**	**Energy Consumed (mAh)**
Circles	16	2.95	2.42	18196
Circles w/ Waypoints	13	0.93	1.31	7301
Follow the Edge	15	0.87	1.53	9689
Greedy	2	5.03	1.52	6521
A*	2	41.44	1.31	4580

Table 2: Results with Path Smoothing Applied

Table 2 shows the results of the paths after the path smoothing method has been applied (Section 4.6.2). Based on the findings in Table 2, path smoothing has a small effect on computation time, adding between 0.73 seconds and 2.26 seconds, but can bring the distance travelled closer to the straight line distance by up to 12%, in the case of *Obstacle Avoidance using Circles*.

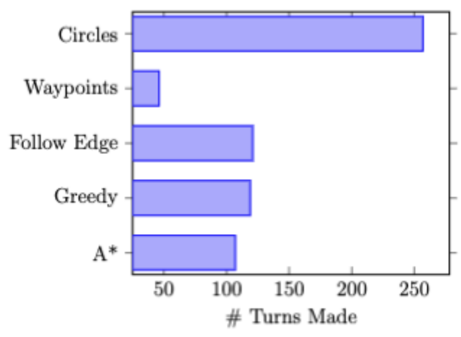
Follow the Edge seems to be the algorithm least affected by path smoothing. This is likely because the algorithm generally returns paths that follow the edge of the obstacle at the maximum altitude, while the path smoothing algorithm (9) only removes waypoints that are below the flying altitude.

Any results used in Sections 5.2.1, 5.2.2, 5.2.3 all use the figures with path smoothing applied.

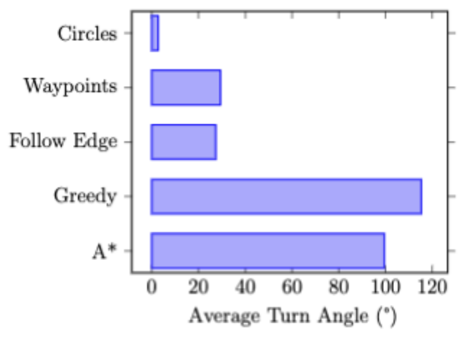
5.2.1 Path Returned - Turns

An important factor in determining the efficiency of a path between two points is the number of turns the drone must make, but also the average angle of each of these turns [23]. *Obstacle Avoidance using Circles* is expected to return paths with a high number of turns but a low average angle, while *Obstacle Avoidance using Waypoints within Circles* should have fewer turns and a higher average angle.

Figure 16: Two Bar Charts about Angles on Paths Returned



(a)



(b)

From *Figure 16*, *Obstacle Avoidance using Waypoints within Circles* is clearly the optimal algorithm in terms of minimising not only the number of turns but also the average angle of the turns. *Obstacle Avoidance using Circles* behaves as expected, with by far the most turns and by far the lowest average turning angle. The fact that A* is not only worse than *Obstacle Avoidance using Waypoints within Circles*, but also worse than *Follow the Edge* at minimising the number of turns and average turn angle is the most surprising part of this graph.

This is possibly due to the 25 points generated requiring a lot of sharp turns to find a path from the start to the end point, as the *Greedy* algorithm also has a large average angle value. Another possible reason is the high fail rate of the other three algorithms. The issue described above where both circles algorithms may receive the straight line path could also be the reason the turn and angle numbers are so low for *Obstacle Avoidance using Waypoints within Circles*, as the straight line path has zero turns and zero turning angle. However, this didn't seem to affect the values returned by

Obstacle Avoidance using Circles, which has an even higher fail rate than it's variant, so this error may not be the root cause.

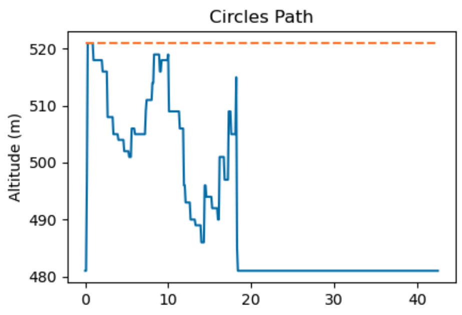
5.2.2 Paths Returned - Metres Ascending and Descending

Another metric that is key to measuring the optimality of a given path is the number of metres ascended and descended. The Amazon crash leading to a brush fire was started after the drone failed when changing it's trajectory from an upward trajectory to straight ahead [50], so minimising this factor can lead to safer paths. To draw the below graphs, each algorithm was given three pairs of points. The drone's flying altitude is set to 80 metres above the altitude of the *start* point, with the maximum altitude set to 120 metres above the *start* point's altitude. Therefore, a drone will change it's altitude for any points that are in between these two points. A graph visualising the altitude over the entire journey from start to end will be useful to compare each algorithm.

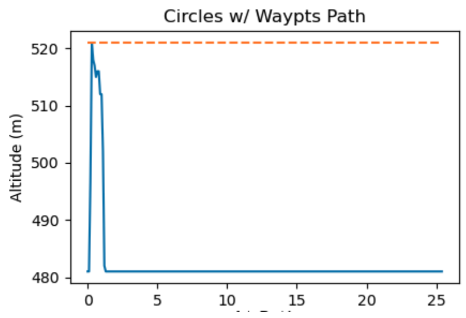
Looking at the first set of points in *Figure 17*, *A** is far ahead of all other algorithms, with *Obstacle Avoidance using Circles* in particular ascending and descending for almost half of the journey as well as flying more than 15 kilometres more. The other three algorithms don't perform quite as poorly, with the paths looking similar to each other based on this graph, spiking at the start of the graph and then becoming flat until the end. *A** has a much smaller spike meaning the drone must ascend and descend much fewer metres while covering no more distance than the other paths.

The graphs in *Figure 18* also show some interesting data. The path returned by the *Follow the Edge* algorithm violates the maximum altitude by around 20 metres. This is likely due to the issue described in the Implementation section (Section 4.5.2), where the drone may need to fly back towards the *start* point to navigate the obstacle. Also worth noting is the very spiky graph produced by the *Greedy* path. This is likely due to the drone following the edge of an obstacle and hence constantly having to change it's flying altitude. At first glance, the *A** path might seem to be outperformed by both of the algorithms that use circles, but looking at the distance travelled reveals that while the *A** path may have to spend more time ascending and descending, it will be flying for a much shorter distance.

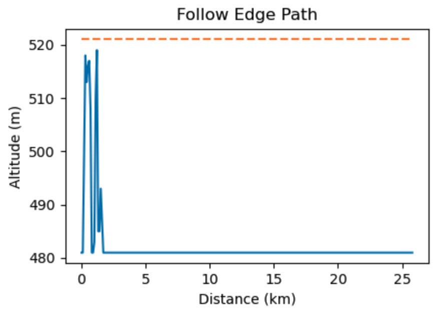
Figure 17: Elevation Change across Path - First Pair of Points



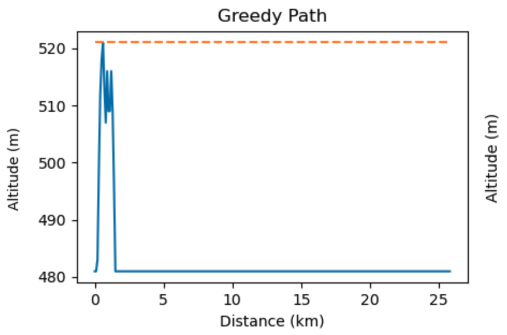
(a)



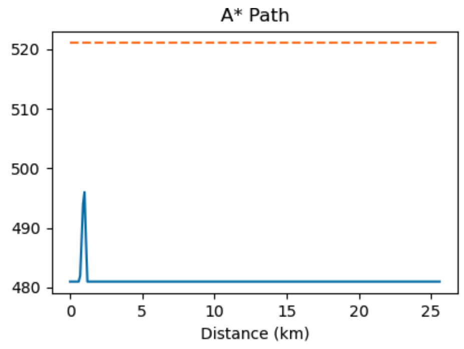
(b)



(c)

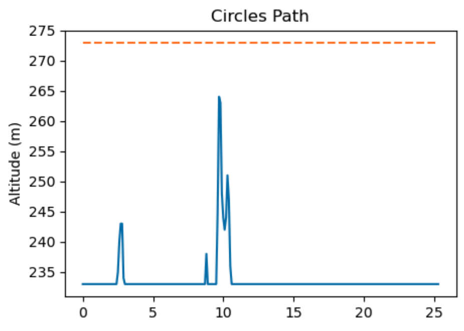


(d)

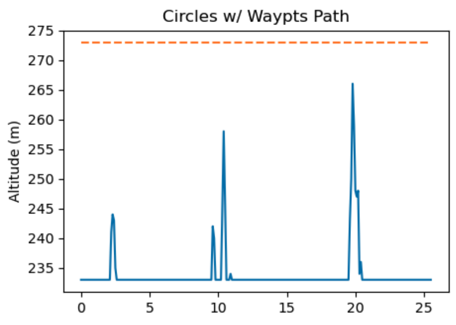


(e)

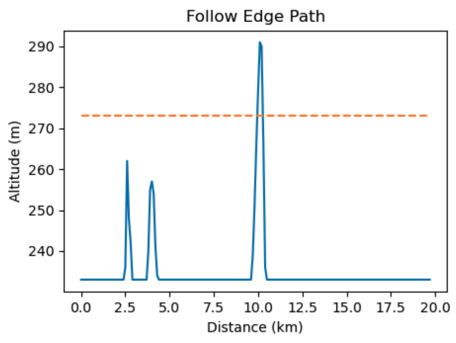
Figure 18: Elevation Change across Path - Second Pair of Points



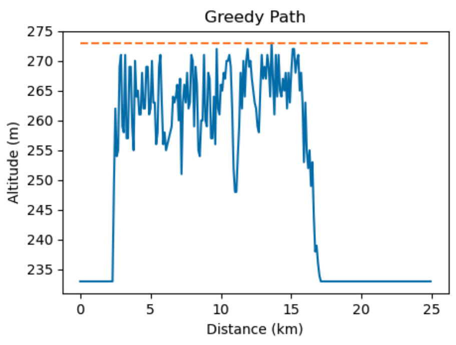
(a)



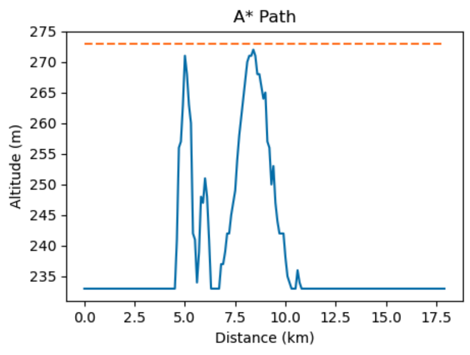
(b)



(c)



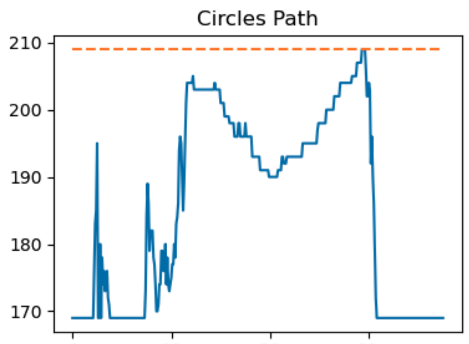
(d)



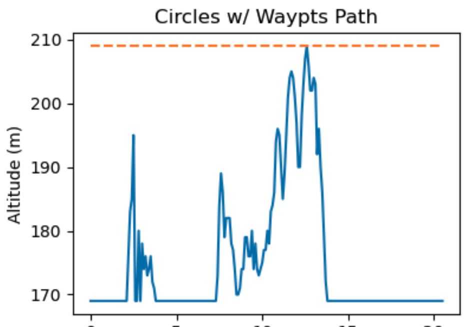
(e)

The path returned by *Follow the Edge* in *Figure 19* is rather interesting. It is completely flat because the algorithm failed to find a path from the *start* point to the *end* point, leading it to search for a path with the maximum altitude value reduced by 40 metres. This results in the path found always staying at or below the flying altitude of the drone which is 80 metres. The *Follow the Edge* path must also fly much longer than any of the other algorithms' paths. The *Obstacle Avoidance using Circles* algorithm returned a path that is more than ten kilometres longer than the path returned by it's variant *Obstacle Avoidance using Waypoints within Circles*. This could be due to the second algorithm removing any circles contained within each other (Section 7). Also surprising is that this path is considerably shorter than not only the *Greedy* path, but also the *A** path, while still remaining under the maximum altitude. The likely cause of this is *A** trying to find a path through the terrain while minimising the number of turns as well.

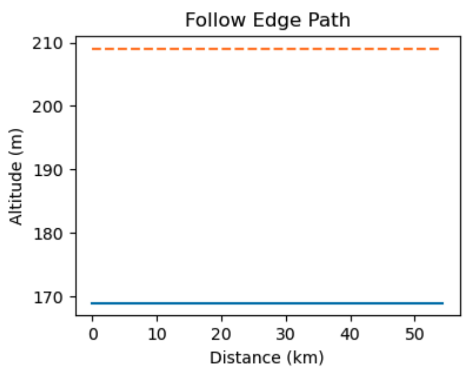
Figure 19: Elevation Change across Path - Third Pair of Points



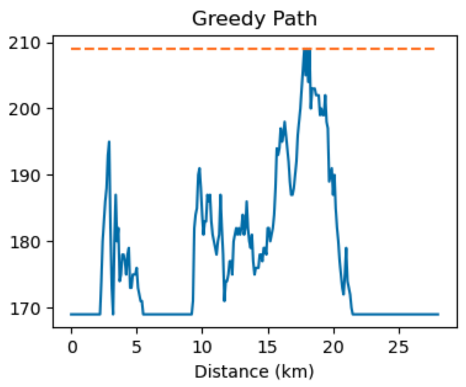
(a)



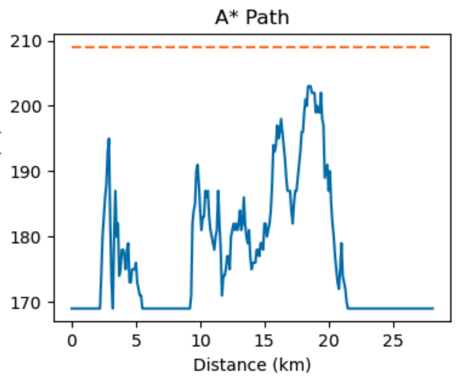
(b)



(c)



(d)



(e)

5.2.3 Paths Returned - Distance Ratio

The next metric used to compare each algorithm is a ratio, the distance between the two points, divided by the total distance covered by the drone in the path returned by the algorithm. This is a key metric to compare each algorithm, as distance is the factor that consumes the most energy based on the energy consumption formula used in this project, where travelling 10 metres more will cost more than ascending 20 metres and then descending down those 20 metres. An extra 10 metres also costs more than the drone making 20 180° turns. The optimal algorithm should travel the least distance, meaning the distance ratio should be the lowest.

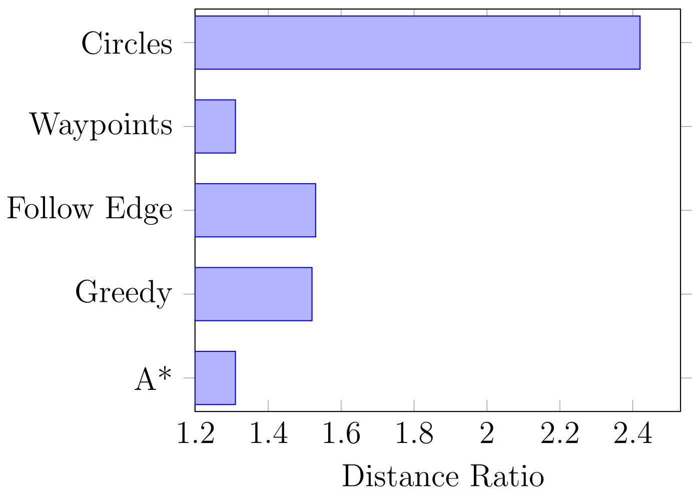


Figure 20: Distance Ratio Comparison

In *Figure 20*, *A** is seen to have the optimal distance ratio (1.31). However, this ratio is also shared by *Obstacle Avoidance using Waypoints*, which is quite surprising as looking at Table 2, the paths returned by this algorithm have a higher energy consumption than *Greedy* with a distance ratio of 1.52. However, this is likely due to the algorithm's high fail rate in comparison to *Greedy*, meaning it is very successful in finding a near-optimal path when it does not fail. The *Obstacle Avoidance using Waypoints* has a very high distance ratio, which is likely due to the algorithm not removing contained circles (Algorithm 7), which can lead to large parts of the path being retraced.

5.3 Requirements Satisfied

Table 3: Requirements Satisfied

Name	R1	R2	R3	R4	R5	R6	R7
Circles		✓			✓		✓
Circles w/ Waypoints		✓			✓		✓
Follow the Edge		✓			✓		✓
Greedy		✓			✓		✓
A*	✓	✓		✓	✓		✓
Bisect Obstacles		✓			✓		✓

A table summarising the algorithms with the requirements (Section 2.5) they have met can be seen in *Table 3.* Requirements 3:

The program must have an interface to input details, such as start and end points, drone information (battery, maximum height, strength, etc.), and topographic maps.

and 6:

The program could allow the user to define the preferred path, e.g. shortest path (distance covered), quickest path, most battery-efficient path.

have not been fulfilled by any algorithm, but this is simply due to the focus of the project shifting away from the original requirements. A graphical user interface could be implemented to fulfill these requirements, but it was not deemed necessary as the optimal path became a combination of the factors mentioned in requirement 6, and the details mentioned in requirement 3 can be tweaked within the code.

As seen in Table 3, *A** not only returns the optimal path out of all algorithms, it all satisfies the most requirements of the project, by taking into account the energy consumption of the UAV.

6 Conclusion

6.1 Project Summary

Over the course of this project, several algorithms were implemented and thoroughly tested on a variety of *start* and *end* points in order to find an optimal path between the two points with respect to energy consumption. The *Obstacle Avoidance using Circles* algorithm returned the paths that consumed the most energy, while the *Obstacle Avoidance using Waypoints* paths were very close to optimal in terms of distance covered and minimising the number of turns. However the algorithm failed in 14 out of 25 tests, so only excels in a specific type of terrain. The *A** algorithm returned the optimal path on every run, minimising the number of turns, the number of metres ascended and descended, and the distance travelled, leading this to be the preferred algorithm of those presented in this project, despite the high computational time.

This study is not without its limitations. The types of terrain in which each algorithm succeeds and fails could be explored further, which may lead to a better understanding of each methods advantages and shortcomings, which could be fed back into the algorithms to combat these shortcomings.

While many current research papers implement algorithms to find optimal paths with respect to one or two of the factors path length, number of turns and number of metres ascended and descended, this project places more of a focus minimising these three factors. Modifying A* to include these two factors leads to an increase in the computational time, but also leads to more energy efficient paths, with the paths returned requiring fewer unnecessary maneuvers for the UAV to make.

6.2 Future Work

Future work should design for and implement the *Bisect Obstacles* algorithm to determine its effectiveness and efficiency.

There are also several more graph-search based algorithms that could be implemented to compare to A*, such as Theta*, Lazy Theta*, Dynamic A* (D*), Anytime D*. Theta* and Lazy Theta* could possibly reduce the path length, but are unlikely to reduce the computational time compared to the modified A* implemented. Meanwhile D* and Anytime D* are unlikely to reduce the path length, but could resolve the issue with computational time for the modified A*.

Another step could be to try and reduce the computational time of A* by changing the inputted graph. One possible way to do this would be to reduce the grid size when the terrain is not so complex. In other words if there is no obstacle above the flying altitude, then the distance between nodes or points could be more than the default 100 metres, which will reduce the search space of A*.

Bibliography

[1] Hasini Viranga Abeywickrama et al. "Comprehensive Energy Consumption Model for Unmanned Aerial Vehicles, Based on Empirical Studies of Battery Performance". In: *IEEE Access* 6 (2018), pp. 58383–58394. DOI: 10.1109/ACCESS.2018.2875040.

[2] Faiyaz Ahmed et al. "Recent Advances in Unmanned Aerial Vehicles: A Review". In: *Arabian Journal for Science and Engineering* 47.7 (July 1, 2022), pp. 7963–7984. DOI: 10.1007/s13369-022-06738-0. URL: https://doi.org/10.1007/s13369-022-06738-0.

[3] Kamil A. Alotaibi et al. "Unmanned aerial vehicle routing in the presence of threats". In: *Computers & Industrial Engineering* 115 (2018), pp. 190–205. ISSN: 0360-8352. DOI: https://doi.org/10.1016/j.cie.2017.10.030. URL: https://www.sciencedirect.com/science/article/pii/S036083521730517X.

[4] Civil Aviation Authority. *The Drone and Model Aircraft Code*. Aug. 2019. URL: https://register-drones.caa.co.uk/drone-code/where-you-can-fly (visited on 04/2023).

[5] Rachel Becker. *I launched a blood-delivering drone*. Apr. 13, 2018. URL: https://www.theverge.com/2018/4/13/17206398/zipline-drones-delivery-blood-emergency-medical-supplies-startup-rwanda-tanzania (visited on 04/2023).

[6] Nicholas Carlson. *Bezos On Amazon's Delivery Drones: 'This Looks Like Science Fiction. It's Not.* Dec. 2, 2013. URL: https://www.businessinsider.com/jeff-bezos-on-amazon-delivery-drones-2013-12?r=US&IR=T (visited on 04/2023).

[7] Vivian Chiciudean and Florin Oniga. "Pathfinding in a 3D Grid for UAV Navigation". In: *2022 IEEE 18th International Conference on Intelligent Computer Communication and Processing (ICCP)*. 2022, pp. 305–311. DOI: 10.1109/ICCP56966.2022.10053965.

[8] CNBC. *Amazon Prime Air drone business stymied by regulations, weak demand*. Mar. 11, 2023. URL: https://www.cnbc.com/2023/03/11/amazon-prime-air-drone-business-stymied-by-regulations-weak-demand.html (visited on 04/2023).

[9] Andrea Cornell. *Drones take to the sky, potentially disrupting last-mile delivery*. Jan. 3, 2023. URL: `https://www.eorc.jaxa.jp/ALOS/en/dataset/aw3d_e.htm` (visited on 04/2023).

[10] Jack Daleo. *Alphabet's Wing launches commercial drone delivery in Texas*. Apr. 7, 2022. URL: `https://www.freightwaves.com/news/alphabets-wing-launches-commercial-drone-delivery-in-texas` (visited on 04/2023).

[11] Kevin Danancier et al. "Comparison of Path Planning Algorithms for an Unmanned Aerial Vehicle Deployment Under Threats". In: *IFAC-PapersOnLine* 52.13 (2019). 9th IFAC Conference on Manufacturing Modelling, Management and Control MIM 2019, pp. 1978–1983. ISSN: 2405-8963. DOI: `https://doi.org/10.1016/j.ifacol.2019.11.493`. URL: `https://www.sciencedirect.com/science/article/pii/S2405896319314776`.

[12] Kenny Daniel et al. "Theta*: Any-Angle Path Planning on Grids". In: *J. Artif. Intell. Res. (JAIR)* 39 (Jan. 2014). DOI: `10.1613/jair.2994`.

[13] Manna Drone Delivery. *Manna - What We Deliver*. URL: `https://www.manna.aero/#OrderAnything` (visited on 04/2023).

[14] DJI. *DJI Matrice 100 Specifications*. Apr. 2023. URL: `https://www.dji.com/ie/matrice100/info#` (visited on 04/2023).

[15] Marco Dorigo, Mauro Birattari, and Thomas Stützle. "Ant Colony Optimization: Artificial Ants as a Computational Intelligence Technique". In: *IEEE Computational Intelligence Magazine* 1 (Jan. 2006), pp. 28–39. DOI: `10.1109/CI-M.2006.248054`.

[16] Edward Fu. *A First-Ever Look at the Sustainability of Autonomous Aerial Logistics*. Apr. 21, 2021. URL: `https://www.flyzipline.com/articles/a-first-ever-look-at-the-sustainability-of-autonomous-aerial-logistics` (visited on 04/2023).

[17] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. DOI: `10.1109/TSSC.1968.300136`.

[18] Sean Hollister. *Amazon's delivery drones served fewer than 10 houses in their first month*. Feb. 2, 2023. URL: https://www.theverge.com/2023/2/2/23582294/amazon-prime-air-drone-delivery (visited on 04/2023).

[19] Drone Industry Insights. *Drone Delivery Market Report 2019-2024*. Nov. 5, 2019. URL: https://droneii.com/product/drone-delivery-market-report-2019 (visited on 04/2023).

[20] Drone Industry Insights. *Top Drone Applications*. Apr. 27, 2022. URL: https://droneii.com/top-drone-applications (visited on 04/2023).

[21] Enterprise Ireland. *MANNA puts plans in motion as the drone delivery service prepares for 2023 expansion*. Aug. 2022. URL: https://www.enterprise-ireland.com/en/News/PressReleases/2022-Press-Releases/MANNA-puts-plans-in-motion-as-the-drone-delivery-service-prepares-for-2023-expansion.html (visited on 04/2023).

[22] Geological Survey Ireland. *Open Topographic Data Viewer*. Apr. 2023. URL: https://dcenr.maps.arcgis.com/apps/webappviewer/index.html?id=b7c4b0e763964070ad69bf8c1572c9f5 (visited on 04/2023).

[23] Mariusz Jacewicz et al. "Quadrotor Model for Energy Consumption Analysis". In: *Energies* 15.19 (2022). ISSN: 1996-1073. DOI: 10.3390/en15197136. URL: https://www.mdpi.com/1996-1073/15/19/7136.

[24] JAXA. *ALOS Dataset*. Apr. 2023. URL: https://www.eorc.jaxa.jp/ALOS/en/dataset/aw3d_e.htm (visited on 04/2023).

[25] Sertac Karaman and Emilio Frazzoli. "Sampling-based algorithms for optimal motion planning". In: *The international journal of robotics research* 30.7 (2011), pp. 846–894.

[26] Sourabh Katoch, Sumit Singh Chauhan, and Vijay Kumar. "A review on genetic algorithm: past, present, and future". In: *Multimedia Tools and Applications* 80.5 (2021), pp. 8091–8126.

[27] L.E. Kavraki et al. "Probabilistic roadmaps for path planning in high-dimensional configuration spaces". In: *IEEE Transactions on Robotics and Automation* 12.4 (1996), pp. 566–580. DOI: 10.1109/70.508439.

[28] Noah Koppel. *Amazon's Recent Patent Raises Fresh Privacy Issues with Drone Delivery*. Oct. 7, 2019. URL: `http://www.fordhamiplj.org/2019/10/07/amazons-recent-patent-raises-fresh-privacy-issues-with-drone-delivery/` (visited on 04/2023).

[29] J.J. Kuffner and S.M. LaValle. "RRT-connect: An efficient approach to single-query path planning". In: *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*. Vol. 2. 2000, 995–1001 vol.2. DOI: `10.1109/ROBOT.2000.844730`.

[30] Steven M. LaValle. *Planning Algorithms*. USA: Cambridge University Press, 2006. ISBN: 0521862051.

[31] Maxim Likhachev et al. "Anytime Dynamic A*: An Anytime, Replanning Algorithm". In: *Proceedings of the Fifteenth International Conference on International Conference on Automated Planning and Scheduling*. ICAPS'05. Monterey, California, USA: AAAI Press, 2005, pp. 262–271. ISBN: 1577352203.

[32] Yucong Lin and Srikanth Saripalli. "Sampling-Based Path Planning for UAV Collision Avoidance". In: *IEEE Transactions on Intelligent Transportation Systems* 18.11 (2017), pp. 3179–3192. DOI: `10.1109/TITS.2017.2673778`.

[33] Katherine Long. *Amazon has gutted the safety teams for its ambitious drone delivery program, as employees warn of stepped-up pressure to meet delivery goals*. Feb. 1, 2023. URL: `https://www.businessinsider.com/amazon-prime-air-safety-teams-drone-delivery-layoffs-2023-2?r=US&IR=T` (visited on 04/2023).

[34] Katie Brigham Lora Kolodny. *Zipline unveils P2 delivery drones that dock and recharge autonomously*. Mar. 15, 2023. URL: `https://www.cnbc.com/2023/03/15/zipline-unveils-p2-delivery-drones-that-dock-and-recharge-autonomously.html` (visited on 04/2023).

[35] Dilip Mandloi, Rajeev Arya, and Ajit K. Verma. "Unmanned aerial vehicle path planning based on A* algorithm and its variants in 3d environment". In: *International Journal of System Assurance Engineering and Management* 12.5 (2021), pp. 990–1000.

[36] Manna. *Manna FAQ*. URL: `https://www.manna.aero/faq` (visited on 04/2023).

[37] Ellips Masehian and Golnaz Habibi. "Robot Path Planning in 3D Space Using Binary Integer Programming". In: 1 (Jan. 2007).

[38] Mathworks. *Probabilistic Roadmaps (PRM)*. URL: `https://uk.mathworks.com/help/robotics/ug/probabilistic-roadmaps-prm.html` (visited on 04/2023).

[39] Shuttle Radar Topography Mission. *Open Topographic Data Viewer*. Sept. 2014. URL: `https://www2.jpl.nasa.gov/srtm/` (visited on 04/2023).

[40] Syed Agha Hassnain Mohsan et al. "Towards the unmanned aerial vehicles (UAVs): A comprehensive review". In: *Drones* 6.6 (2022), p. 147.

[41] Alex Nash, Sven Koenig, and Craig Tovey. "Lazy Theta*: Any-Angle Path Planning and Path Length Analysis in 3D." In: vol. 1. Jan. 2010.

[42] Ciara O'Brien. *Coca-Cola HBC backs Irish drone delivery start-up Manna*. Mar. 16, 2023. URL: `https://www.irishtimes.com/business/2023/03/16/coca-cola-hbc-backs-irish-drone-delivery-start-up-manna/` (visited on 04/2023).

[43] Garth Paine. *Have You Heard the Buzz About Delivery Drones' Noise?* May 6, 2019. URL: `https://www.aviationtoday.com/2021/04/06/zipline-expands-japan-toyota-partnership/` (visited on 04/2023).

[44] Prashant Pandey, Anupam Shukla, and Ritu Tiwari. "Three-dimensional path planning for unmanned aerial vehicles using glowworm swarm optimization algorithm". In: *International Journal of System Assurance Engineering and Management* 9.4 (Aug. 1, 2018), pp. 836–852. DOI: `10.1007/s13198-017-0663-z`. URL: `https://doi.org/10.1007/s13198-017-0663-z`.

[45] Jon Porter. *Amazon's Prime Air inches closer to takeoff in the US with FAA approval*. Aug. 31, 2020. URL: `https://www.theverge.com/2020/8/31/21408646/amazon-prime-air-drone-delivery-faa-clearance-approval-health-safety-alphabet-wing` (visited on 04/2023).

[46] Lun Quan et al. "Survey of UAV motion planning". In: *IET Cyber-Systems and Robotics* 2.1 (2020), pp. 14–21. DOI: https://doi.org/10.1049/iet-csr.2020.0004. eprint: https://ietresearch.onlinelibrary.wiley.com/doi/pdf/10.1049/iet-csr.2020.0004. URL: https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/iet-csr.2020.0004.

[47] Kelsey Reichmann. *Zipline Expands into Japan with Toyota Partnership*. Apr. 6, 2021. URL: https://slate.com/technology/2019/05/delivery-drones-amazon-google-noise-buzzing.html (visited on 04/2023).

[48] Vincent Roberge, Mohammed Tarbouchi, and Gilles Labonte. "Comparison of Parallel Genetic Algorithm and Particle Swarm Optimization for Real-Time UAV Path Planning". In: *IEEE Transactions on Industrial Informatics* 9.1 (2013), pp. 132–141. DOI: 10.1109/TII.2012.2198665.

[49] Thiago Rodrigues et al. "Drone flight data reveal energy and greenhouse gas emissions savings for small package delivery". In: (Nov. 2021).

[50] Emma Roth. *Amazon is still struggling to make drone deliveries work*. Apr. 12, 2022. URL: https://www.theverge.com/2022/4/11/23020549/amazon-struggling-drone-deliveries-prime-air-bezos (visited on 04/2023).

[51] Balthazar Rouberol. *haversine Documentation*. Apr. 2023. URL: https://pypi.org/project/haversine/ (visited on 04/2023).

[52] Manuel Schmitt and Rolf Wanka. "Particle swarm optimization almost surely finds local optima". In: *Theoretical Computer Science* 561 (2015). Genetic and Evolutionary Computation, pp. 57–72. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2014.05.017. URL: https://www.sciencedirect.com/science/article/pii/S0304397514004150.

[53] Umar Shakir. *Zipline's new drones release tethered mini-drones for precision package deliveries*. Mar. 15, 2023. URL: https://www.theverge.com/2023/3/15/23639425/zipline-drone-delivery-autonomous-tether-droid (visited on 04/2023).

[54] Jamie Skykam. *Drone Statistics*. Dec. 2022. URL: https://skykam.co.uk/drone-statistics/ (visited on 04/2023).

[55] Amazon Staff. *Amazon Prime Air prepares for drone deliveries.* June 13, 2022. URL: https://www.aboutamazon.com/news/transportation/amazon-prime-air-prepares-for-drone-deliveries (visited on 04/2023).

[56] Amila Thibbotuwawa et al. "Energy Consumption in Unmanned Aerial Vehicles: A Review of Energy Consumption Models and Their Relation to the UAV Routing". In: *Information Systems Architecture and Technology: Proceedings of 39th International Conference on Information Systems Architecture and Technology – ISAT 2018.* Cham: Springer International Publishing, 2019, pp. 173–184. ISBN: 978-3-319-99996-8.

[57] UPS. *Drone COVID Vaccine Deliveries.* Aug. 23, 2022. URL: https://about.ups.com/ca/en/our-stories/innovation-driven/drone-covid-vaccine-deliveries.html (visited on 04/2023).

[58] Frank Warmerdam. *GDAL Documentation.* Apr. 2023. URL: https://gdal.org/ (visited on 04/2023).

[59] Qian Xue, Peng Cheng, and Nong Cheng. "Offline path planning and online replanning of UAVs in complex terrain". In: *Proceedings of 2014 IEEE Chinese Guidance, Navigation and Control Conference.* 2014, pp. 2287–2292. DOI: 10.1109/CGNCC.2014.7007525.

[60] Fei Yan, Yi-Sha Liu, and Ji-Zhong Xiao. "Path Planning in Complex 3D Environments Using a Probabilistic Roadmap Method". In: *International Journal of Automation and Computing* 10.6 (2013), pp. 525–533.

[61] Yong Zeng and Rui Zhang. "Energy-Efficient UAV Communication With Trajectory Optimization". In: *IEEE Transactions on Wireless Communications* 16.6 (2017), pp. 3747–3760. DOI: 10.1109/TWC.2017.2688328.

[62] Chao Zhang et al. "UAV path planning method based on ant colony optimization". In: *2010 Chinese Control and Decision Conference.* 2010, pp. 3790–3792. DOI: 10.1109/CCDC.2010.5498477.

[63] Juan Zhang et al. "Energy consumption models for delivery drones: A comparison and assessment". In: *Transportation Research Part D: Transport and Environment* 90 (2021), p. 102668. ISSN: 1361-9209. DOI: https://doi.org/10.1016/j.trd.2020.102668. URL: https://www.sciencedirect.com/science/article/pii/S1361920920308531.

[64] Zipline. *Zipline Technology*. Apr. 2023. URL: `https://www.flyzipline.com/technology` (visited on 04/2023).