Informatics Large Practical Report

Section 1: Software Architecture Description

The application is made up of 9 main classes, App, CentralArea, Delivery, DroneMove, Item, LngLat, NoFlyZone, Order and Restaurant. The application also includes an InvalidPizzaCombinationException class as well as an OrderOutcome Enum.

App

The main class is the App class. This is the entry point of the application through which all user input is given, I/O error handling is performed, all required data is accessed from the REST server, orders are processed, the drone control algorithm is executed, the flight path is created and all output files are created.

This class has no attributes and contains several methods to perform the functionality detailed above. The most important method is the main method, which ties all of the other methods within the class and consequently within the whole program together. Main reads in user input from the command line and checks if enough arguments have been passed in. Then the date and base URL are set and Main executes the other methods within the class.

The first non-main method the class contains is the checkURLs method which verifies that all REST requests performed by the program are valid. This means the URLs exist and are in the correct format, and the data returned is of the type expected. This method exists to prevent errors from arising for the first time within the main execution of the program. This allows for more useful error messages and reduces the possibility of a null pointer exception and other irritating errors occurring during the program’s runtime.

Then for the drone control algorithm, the App class contains the following methods: findBestMove which returns the best legal non-repeated drone move at a given position, makeOrderPathToGoal which generates a one-way section of the drone’s flight path to a specified goal and makeFullOrderPath which returns the full flight path of a drone delivering a single order. Finally, this class contains the method writeFiles which writes the 3 required output files for the inputted date: drone-YYYY-MM-DD.geojson, deliveries-YYYY-MM-DD.json and flight path-YYYY-MM-DD.json. Each of these files is overwritten by the class if they already exist otherwise, they are created anew.

This class is required as there needs to be a class which brings together all the functionality of all other classes within the program.

CentralArea

The next class is the singleton CentralArea class, which stores the four points of the rectangular central area. The main method of this class is getPoints which simply returns the vertices of the central area, to be used in determining whether a point is within this area. The other method of this class is getInstance which is used for access to the singleton class instance.

This class is a singleton because when it is used by the LngLat class, an instance of the class is used, which is retrieved from the REST server. However, if a new instance of the class is used every time, firstly network I/O is very slow in comparison to other aspects of the program, so accessing a new instance would significantly slow down the runtime of the program. Moreover, it would be possible that the points of the central area would change over the runtime, or even that the REST server may fail when retrieving later instances of the class. So the CentralArea class is singleton to ensure that the value of the class is constant through the runtime of the program and the REST request need only be made once.

Delivery

Another class is the Delivery class. This class stores the information about each order that is to be written to the deliveries-YYYY-MM-DD.json output file, that is the order number, the cost of the order and the outcome of that order.

The main method in this class is the getDeliveries method, which takes in all orders on a certain day and all the restaurants and returns a Delivery object for each order which includes whether the order is valid and if not, the reason why. This then allows the drone control algorithm to only need to consider valid orders, as the validity is determined beforehand.

The other two main methods in the class are used after the drone control algorithm is executed, as they are setting the outcome of the order to either Delivered or ValidButNotDelivered. These methods are needed because it won't be known before the execution of the algorithm which of these 2 values will be taken.

This class was chosen as the use of a custom class for data binding allows the deliveries output file to be easily written to when using Objectmapper directly, by using the Delivery class. Moreover, the use of the Delivery class allows the immutability of the Order class, as any information about the order which is decided during the runtime of the program is stored in Delivery.

DroneMove

The DroneMove class is used to represent an individual move made by the drone, in the format specified for the flight path-YYYY-MM-DD.json file.

The main methods in this class are simply constructors. The class has 2 constructors, one in which the flight angle is given and one where the flight angle is not in which case the angle will be 'null', for when the drone hovers to pick up or deliver an order. Another method within this class is makePathCoordinates which turns a list of DroneMove objects (which together represent a flight path) into a list of Point objects defined in Mapbox SDK, for the drone-YYYY-MM-DD.geojson file. The class also has getters, but no setters providing encapsulation [1] and security as the class's attributes are all private and final, resulting in data abstraction to prevent misuse of the class and its attributes.

This class was used firstly to enable easy writing to the required output file using ObjectMapper. Also, the DroneMove class was used to allow easy representation of the flight path of the drone within the program. Sections of the drone's flight path are represented by lists of DroneMove objects. This is to allow easy backtracking during the drone control algorithm as the program can simply iterate through the list in reverse order and make calculations based on the angles in the list. In addition, any information relating to the flight path of the drone will be stored using the DroneMove class, meaning the program will have loose coupling as other classes will have little to do with the representation of the drone's flight path.

Item

Item is a class that represents a single item on offer by a restaurant. It contains the name of the item and its cost in pence. This class is used for clarity, to make it clear when the program is looking through the menu of a restaurant, instead of relying on index lists or arrays which could be confusing. In addition, this class is used to avoid problems with changing data types to represent the name and cost of an item in an array or list. Moreover, this class is a record, as the menu items on offer will not change during the runtime of the program.

LngLat

The LngLat class represents coordinate points on the map, in terms of their longitude and latitude coordinates. It also is used when making calculations in terms of points on the map. The first method in this class is inCentralArea which determines whether a point is within the central area, using an instance of the CentralArea class. LngLat also contains distanceTo which calculates the Euclidean distance between 2 points, closeTo which determines if two points are ‘close’ to each other, that is within 0.00015 degrees of one another and finally nextPosition which calculates the position of the drone after a move of a given number of degrees.

This class was chosen as a focal point through which any calculations involving points on the map are performed. This then allows for both loose coupling, as no other classes will perform calculations involving specific points on the map or will generate points along the drone’s flight path, and high cohesion as all the methods within this class are involved in pointwise geometry.

We also note that this class is also a record. This design choice was made because a set of points within the map, for example the location of the central area or a point along the drone’s flight path, will not change. Because of this fact immutability is an important characteristic of this class.

NoFlyZone

A key class within the program is the NoFlyZone class. This represents an individual part of the no-fly zones that the drone aims to avoid, in the format returned by the REST server. As well as this the class contains two methods relating to determining whether a drone move would cause the drone to enter the no-fly zones.

The first method in this class is getNoFlyZones which retrieves the entirety of the no-fly zones from the REST server and parses this into an array of NoFlyZone objects, then iterates through this array creating a combined Mapbox MultiPolygon object representing all of the no-fly zones together. Then we consider if the drone enters the no-fly zone with the two methods intersectsNoFlyZones and linesIntersect . Firstly, intersectsNoFlyZones calculates whether the path created by the drone intersects any of the perimeters of the no-fly zones. This method uses linesIntersect which calculates if two line segments (given as MapBox SDK LineString objects) intersect each other. The details for this are explained further below in the drone control algorithm section.

This class was deemed necessary because of the line-line intersection problem. Firstly this problem is posed in terms of line segments, not in terms of single points so the methods used would be separate from those in the LngLat class, and no other class handles calculations similar to this, so a separate class is needed to ensure high cohesion within classes. As well as this, considering the line-line intersection problem for every line along the perimeters of multiple polygons resulted in many lines of code which were ultimately split into two methods, needing their own class to aid clarity of code and to prevent classes becoming too cumbersome.

In terms of alternative implementations, simply parsing the data returned from the REST server directly into a Mapbox Polygon object was initially considered however, this was not deemed possible. Firstly the Polygon objects do not allow for names to be parsed in and also require a two-dimensional list to allow for polygons with holes within them, a format different to that returned by the REST request. Thus it was determined that an intermediate class was required, hence why the NoFlyZone class is used; it can parse the data from the REST request directly.

Also, similar to the central area, the no-fly zones will not change during the runtime of the program and will be obtained from the REST server, so a singleton design for the NoFlyZone class was considered. However it was not chosen because the no-fly zones are composed of multiple separate no-fly zone objects, so it made sense to have multiple different instances of a class representing a single no-fly zone. Thus instead the NoFlyZone class was represented as a record, to ensure immutability and consequently data security within the class and its attributes [2].

<u>**Order**</u>

The order class represents an order made, in the format in which it is represented in the REST request. This class has several methods. Firstly, getOrders which performs a REST request to get an array of Order objects for all the orders made on a given day. The class also has the method getValidity, which implements several smaller methods within the class (isCardNumberValid, isCardExpiryValid, isCVVValid, validPizzaCount, pizzasDefined, isPizzaValid, sameSuppliers, correctTotal, getDeliveryCost) that together check each criterion that the order's information must meet to be a valid order. The method getValidity then returns the corresponding Enum for the order outcome if the order is invalid, else it returns that the order is valid. Note that this value will be changed to a value of the order outcome Enum after execution of the drone control algorithm.

Another important method is sortOrders which returns a given list of orders in ascending order sorted by the Euclidean distance from the restaurant the order is picked up from to Appleton tower. This method is executed as part of the drone control algorithm. Finally, this class also has getRestaurant, which identifies the restaurant where the order's items will be picked up from provided the order is valid and getOrderFromOrderNo, which identifies an order object by its order number and is used within sortOrders.

This class is a record, as orders are retrieved once from the server and will not change during the runtime of the program. Hence it was decided that it is important for an order's attributes to be immutable, which guarantees the safety that once data for the order has been accurately parsed in, it will remain safe and valid for the duration of the program whenever this information is used and will not cause unexpected errors during the runtime. As well as this immutability reduces the complexity of the program as there is only a single state of the class during the runtime of the program and ensures the program follows good software design principles, due to the guaranteed safety. It was also considered to give the Order class private attributes with no setters, in order to provide encapsulation. However this functionality was more easily achieved with a record that only allows final attributes.

For this reason, as well, the Delivery class was used, as a class to store mutable attributes relating to specific orders which will necessarily change during runtime whilst allowing a safe and secure immutable representation of the order information retrieved from the REST server.

In addition, all methods which operate on the initial information of the order itself, such as checking the validity of an order before executing the drone control algorithm or sorting the order numbers are held within this class. This means there is a clear focal point in the program for order handling, which would help to debug should any errors arise relating to the orders. Moreover, this aligns with the programming principles of loose coupling, as no other class engages in direct manipulation of an order's data and high cohesion, as all methods within the class relate to either an order and its attributes or a collection of orders.

Restaurant

The Restaurant class represents a restaurant pizzas can be ordered from, in the same format as the data parsed in from the REST server. This is a record, as once the restaurants are retrieved from the REST server they will not change during the entire runtime of the program so the class representing restaurants should be immutable. The only main method in this class is getRestaurantsFromRestServer which sends a REST request to the server and parses this data into an array of Restaurant objects. The Restaurant class is needed to separate the storing of the restaurants and their attributes from other classes.

Section 2: Drone Control Algorithm

The Algorithm employed by the drone is a greedy approach, which after checking which of the orders made have valid information, employs pre-processing through pre-flight path order sorting, then one-way greedy flight path generation followed by flight path backtracking. As for move legality, the algorithm considers 3 aspects: intersection with the no-fly zone, departure from the central area and repetition of moves.

Pre-Flight path sorting

Firstly, all the valid orders made on a certain day are sorted in ascending order, by distance from the starting position of Appleton Tower to the restaurant the pizzas are to be picked up from. This means the orders that are initially closest and would require fewer drone moves are prioritised and delivered first, to greedily increase the number of orders made by the drone within the limited number of moves.

One-way flight path generation

Once the orders have initially been sorted, the algorithm iterates through all of the valid orders in this sorted arrangement. For each order, the drone generates a section of the flight path only going one way; from the initial starting point, which will be close to Appleton Tower, to the restaurant the pizzas in the order have been ordered from.

The algorithm generates this section of the flight path on an individual move-by-move basis, adding the move to the partial flight path before generating the next move. The algorithm calculates the next move by first iterating through all possible angles in which the drone could move. This is all the angles between 0 and 337.5 degrees, in increments of 22.5 degrees, moving clockwise, where an angle of 0 degrees means moving straight north.

Then this move is checked to see if it is legal. The methodology for this is documented below.

If the move is indeed legal, we then check to see if this move is an improvement on the current best move. We do this by storing the value of the current best move at that position and the Euclidean distance between the position after that move and the end goal, which is initialised to be something far greater than the largest feasible value for any move, for example, 1000 which is much larger than 2000 * 0.00015. If the move we are considering is an improvement on this distance we then select that move as the new best move. After all legal moves have been considered, we then add that move to the partial flight path for the current order and consider the best move from the new position. This process is then repeated until the drone is close to its goal of the restaurant, and the drone hovers at the restaurant to pick up the order. After this point, we begin backtracking.

Backtracking

Backtracking involves re-constructing a section of the flight path in reverse order, to return the drone to Appleton Tower. This process is done by iterating through all moves in the flight path, in reverse order, then taking the angle of that move, adding 180 modulo 360 and then calculating the new position of the drone after this move. We then add this reversed move to the partial flight path. As this is backtracking the drone's flight path, we find that the position of the drone after this new move is identical to the position of the drone on the path to the restaurant. As a result, this leads to the drone seemingly 're-trace its steps' very often when delivering pizzas from the same restaurant as the drone often returns to exactly the same point.

Once this backtracking process is finished, the drone will have returned to Appleton Tower where again the drone will hover to drop off the order. After this, we've obtained a complete section of the flight path for the specific order. We consider if adding this section to the current flight path already obtained would cause the drone to have made over 2000 moves on the specified day. If this is the case, we discord this section of the flight path and do not deliver the order, otherwise, we add it to the full flight path of the drone and that order is delivered. We then mark the order as valid but not delivered or delivered accordingly and then repeat this process for the next order.

No-Fly Zone Intersection

A drone move is only valid if it does not cause the path of the drone to intersect the perimeter of any of the areas within the no-fly zone. We check this by first considering each polygon within the no-fly zone. We then split this polygon up into the vertices in the corners of the polygon and then form a set of line segments formed between each adjacent pair of these vertices. This set is then the set of all the straight lines along the perimeter of the polygon.

We then consider whether the line segment formed between the position of the drone before and after the move intersects any of the line segments in the set we have obtained from our polygon.

A solution to this line-line intersection problem is calculated by viewing the two line segments in terms of Bezier parameters [3]. Firstly we represent the line segment between the two points (x_1, y_1) and (x_2, y_2) as

$$L_1 = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} + t \begin{pmatrix} x_2 - x_1 \\ y_2 - y_1 \end{pmatrix}$$

And the second line segment between the two points (x_3, y_3) and (x_4, y_4) as

$$L_2 = \begin{pmatrix} x_3 \\ y_3 \end{pmatrix} + u \begin{pmatrix} x_4 - x_3 \\ y_4 - y_3 \end{pmatrix}$$

We then calculate how far along each line segment the intersection point of the two lines will be by:

$$t = \frac{(x_1 - x_3)(y_3 - y_4) - (y_1 - y_3)(x_3 - x_4)}{(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4)}$$

$$u = \frac{(x_1 - x_3)(y_1 - y_2) - (y_1 - y_3)(x_1 - x_2)}{(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4)}$$

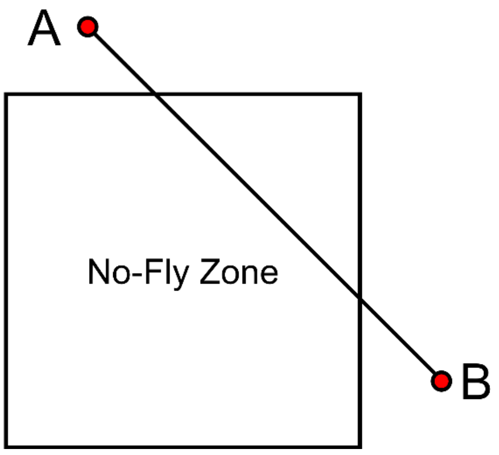

Figure 1: Bad approach - end points not in the No-Fly Zone, but the drone still goes through it

We note that if the denominator of these two terms is 0 then these values will not exist, and so these two lines are parallel and will never intersect, therefore we check the value of the denominator is non-zero first. Then if the denominator is non-zero we calculate each of these values. If they are both between 0 and 1 inclusive then we note that the two line segments do intersect each other. Otherwise, the two line segments will only intersect each other if at least one of them is extended past its start or end point, hence our two finite line segments defined between 2 points do not intersect.

This solution is calculated for each line segment formed by our polygon. If none of these line segments intersects the line segment formed by the drone move then this move stays away from this part of the no-fly zone. We then check this for every polygon within the no-fly zone. If none of these polygons is intersected, the move is valid regarding the no-fly zone.

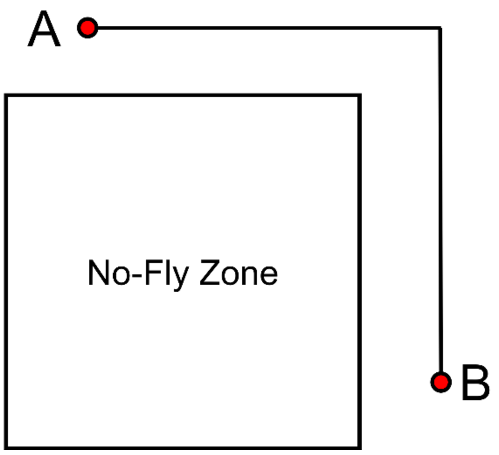

Figure 2: Good approach - Drone fully avoids the No-Fly Zone

Before this approach was taken for intersecting the no-fly zone, an alternative approach was taken where we simply consider whether the endpoint of the line segment formed by a move is in the no-fly zone. However, it was noted that this approach was sometimes incorrect as it would allow moves which enter and then exit the no-fly zone, as shown in Figure 1. So we use the line-line intersection approach which, although more elaborate and less efficient, correctly identifies paths which intersect and conversely that fully avoid the no-fly zone, as shown in Figure 2.

<u>Staying within the central area</u>

The drone is not allowed to move in and out of the central area during the delivery of an order. To stop this the algorithm does not consider moves that would result in the drone leaving and re-entering the central area. As the algorithm uses extensive backtracking, the departure from the central area is considered from the reverse viewpoint – if the drone has left the central area on its way to the restaurant the drone may not then re-enter the central area as backtracking this path would result in leaving and re-entering the central area on the return journey. So only move pairs where the following logical statement is true: if the drone’s position before the move is not in the central area, this implies the drone’s position after the move is not in the central area.

<u>Move repetition</u>

A drone move is considered invalid if it has been repeated on the way to a specific goal. This is because allowing repeated moves may lead to the drone being stuck at the edge of a no-fly zone. We can see this in Figure 3. Once the drone enters point A, it cannot then go through the no-fly zone, so it will enter point B. However once at point B, the point closest to the goal in terms of Euclidean distance will then be point A again, as going around the no-fly zone will temporarily increase the Euclidean distance between the position of the drone and its goal. So the drone will return to point A. We can then see that the drone will simply oscillate between points A and B and will never reach the goal, as the drone is required to go around the no-fly zone.

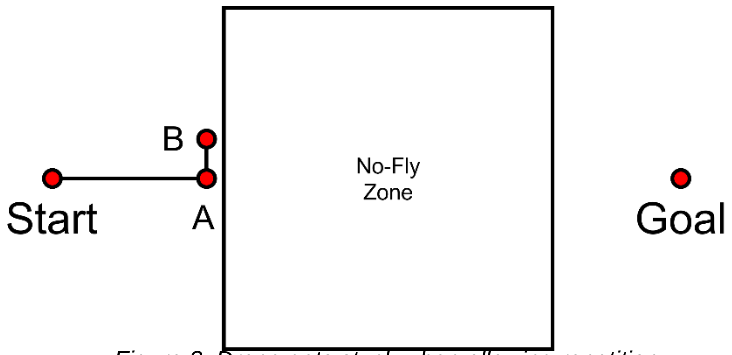

Figure 3: Drone gets stuck when allowing repetition

Thus, we avoid repeating moves. To implement this, at the drone's starting position we initialise a list of points visited by the drone. Then after every move made by the drone, we add this point to the list of visited points. For every move, we discard any moves that would result in the drone re-entering a point already visited (in reality this means within 0.00015 degrees of the point), and only consider moves that would enter new points. So our flight path from Appleton Tower to the restaurant will include no repeated positions. We also note that when we backtrack and reconstruct the path to return to Appleton Tower we allow positions to be repeated from the original flight path to the restaurant, and observe that this section of the flight path when returning will also have no repeated points. Then when we have returned to Appleton Tower the list of repeated points will be cleared, as we only consider moves to be repeated if they are repeated whilst attempting to pick up the same order.

Improvements

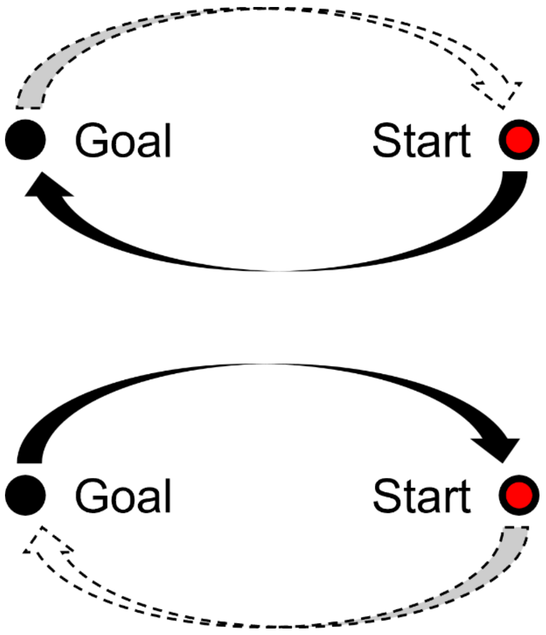

Figure 4 - Consider the flight path 2 different ways: construct the black direction path first and backtrack this path in the dashed direction

One improvement that could be made to the algorithm to improve the optimality of the drone's flight path could be to consider the reversed approach to the flight path algorithm as well. That is, consider the flight path that is created by the greedy move-by-move algorithm in reverse order from the path the drone will actually take when delivering the order; starting from the restaurant and travelling towards the drop-off point. Then backtrack this path from the drop-off point to the restaurant. Then swap the first and second halves of the flight path, to create a new flight path that may be different to the one created by starting the greedy flight path algorithm at the drop-off point.

The two approaches are illustrated in Figure 4 and the drone would take whichever of the two flight paths is the shortest. This new approach of taking the best of 2 paths would mean the result of the flight path would be at worst the same as the original approach and potentially better, producing a shorter flight path which would allow more orders to be delivered.

A major flaw with this greedy approach is that it is not guaranteed to be an optimal solution. Seeing as the flight path must go around no-fly zones the algorithm could be drawn to moving in a direction where the drone originally moves closer to the goal but must then take a longer route overall, compared to a path where the drone initially moves away from its goal but then takes a more direct path. To this end, a provably optimal approach such as the A* algorithm [4] was considered, however, this approach has a significant drawback in terms of memory required. As there are potentially 16 possible moves at each point (including repetitions) and paths to and from a restaurant will very often exceed a length of 100 moves the number of nodes that would need to be stored in memory was deemed infeasible as the memory requirement for A* is $O(b^d)$CITATION Rus098 \l 1033 [5], where b is the average number of moves available at a point and d is the depth of moves in a solution. However some form of graph pre-processing before any moves are calculated or an implementation of a memory-bounded algorithm, such as SMA* [6] might allow for a path to be created where there is a guarantee of optimality, with feasible memory requirements. Although again this could have a potential drawback of presenting a long and complicated algorithm which would be difficult to understand and debug and would potentially hinder code clarity heavily.

<u>**Figures**</u>

The algorithm was tested on every day within the range of dates that provided order data. The effect of sorting can be seen clearly as the drone never delivers orders for pizzas at the farthest restaurant from Appleton Tower, Sora Lella. The algorithm prioritises closer orders to greedily increase the number of pizzas that could be delivered before the drone's battery runs out. Moreover, we can see the effect of backtracking – the drone returns to Appleton Tower taking the exact same flight path as it took travelling to the restaurant.

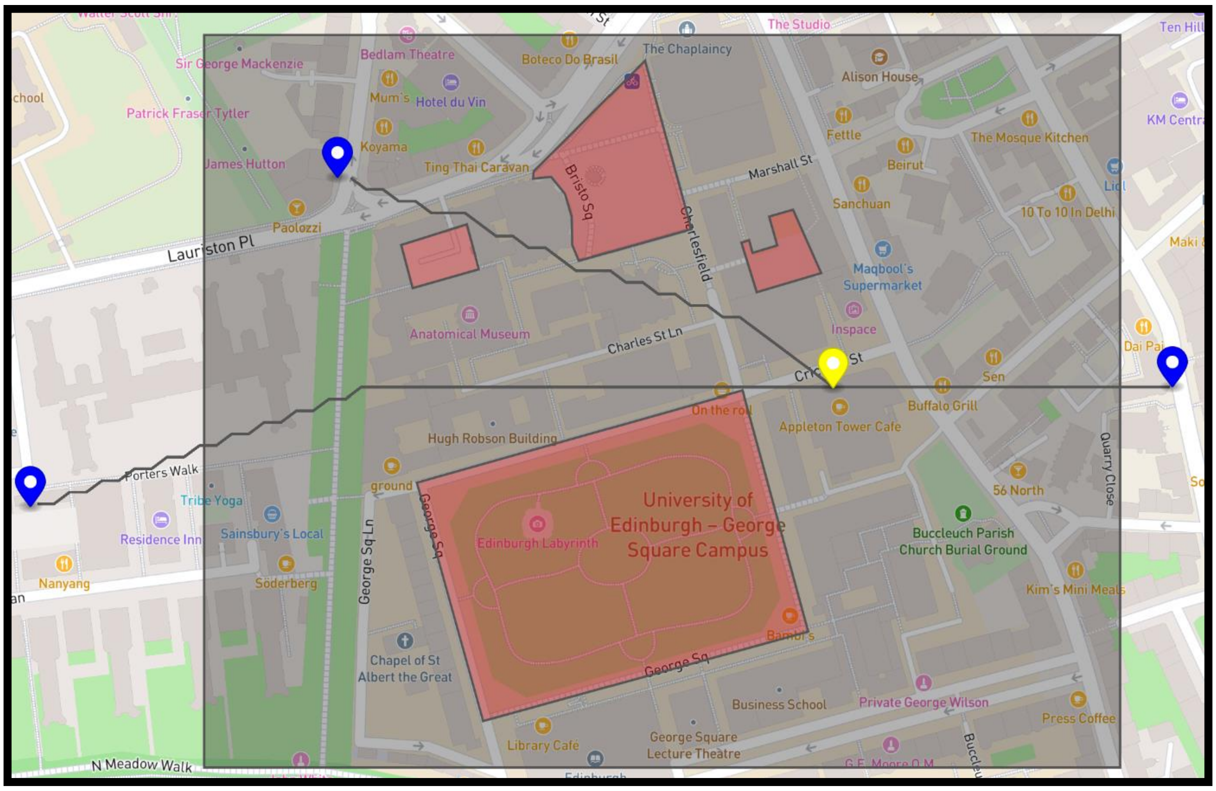

Figure 5 - Flight path on 2023-02-19

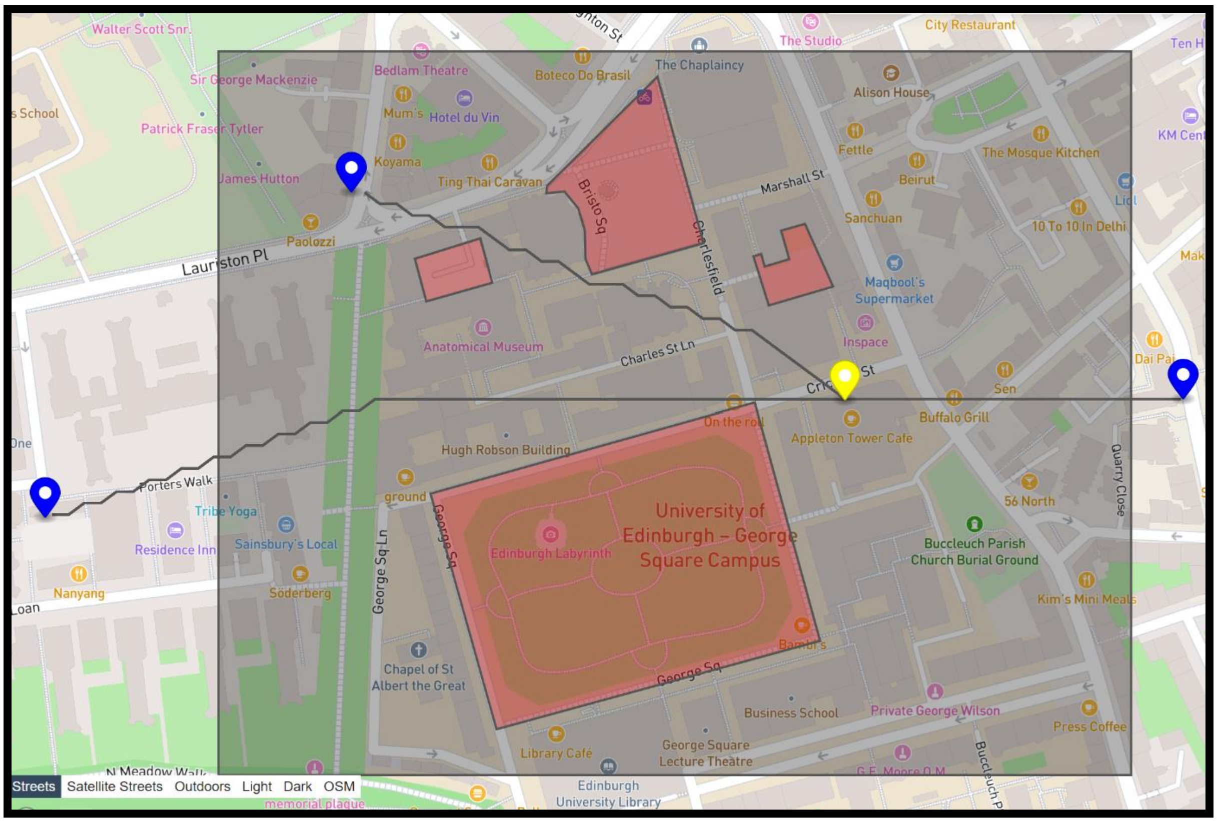
University of Edinburgh – George Square Campus
Lauriston Pl
Bristo Sq
Charlesfield
Charles St Ln
Marshall St
George Sq
George Sq Ln
Porters Walk
Streets Satellite Streets Outdoors Light Dark OSM

Figure 6 - Flight path on 2023-04-15

References

[1] J. Mitchell, "Concepts in Programming Languages," 2002, p. 418.

[2] B. Goetz and e. al, "Java Concurrency in Practice," 2006, pp. 46-49.

[3] Wikipedia, "Line-Line Intersection," 21 11 2022. [Online]. Available: https://en.wikipedia.org/wiki/Line%E2%80%93line_intersection#Given_two_points_on_each_line_segment.

[4] S. Russell and P. Norvig, "Artificial Intelligence: A Modern Approach 3rd Edition," 2009, p. 93.

[5] S. Russell and P. Norvig, "Artificial Intelligence: A Modern Approach 3rd Edition," 2009, p. 98.

[6] S. Russell and P. Norvig, "Artificial Intelligence: A Modern Approach 3rd Edition," 2009, p. 101.