Informatics Large Practical Report

James Ward – s2012077

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 flightpath is created and all output files are created.

Firstly user input is read from the command line by the App class. This input is then checked, so enough arguments have been passed in for the program to work. Then the date and URL for the REST server are set and this class then checks that all URLs required for the program to run exist, are in a valid format and that any REST request used can be made and returns data in the format expected. If would occur from accessing the REST server, they will be caught at this stage. Then all orders for the given day, all restaurants that pizzas can be ordered from and the No-Fly Zones are retrieved from the server.

[more detail about methods for DCA?]

Next, the drone control algorithm is implemented, which is given in more detail in the next section. After the algorithm has run, the delivery statuses of each order made will have been set. The App class then writes to the 3 output files required for that date: drone-[date].geojson, deliveries-[date].json and flightpath-[date].json. Each of these files is overwritten by the class if they already exist, otherwise they are created anew.

CentralArea

The next class is the singleton CentralArea class. This represents 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 a points position within this area. The other method of this class is the getInstance() class which is used for singleton access to it.

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, 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 during 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 represents the information about each order that is to be written to the deliveries-[date].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 this point which of these 2 values will be taken.

This class was chosen as it allows the deliveries output file to be easily written to, when using Objectmapper as any and all relevant data is within the 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 flightpath-[date].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.

This class was used firstly to enable easy writing to the required output fie using ObjectMapper. Also this class was used to allow easy representation of the flightpath of the drone within the program. Sections of the drone's flightpath 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 upon the angles in the list. In addition, any information relating to the flightpath 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 representation of the drone's flightpath.

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, as well as 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 of the map. The main methods in this class are inCentralArea() which determines whether a point is within the central area, 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 point at which the drone will be after a move of a given number of degrees.

This class is used as it is 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 to generate points along the drone's flightpath, 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 as a set points within the map, for example the location of the central area or a point along the drone's flightpath, will not change hence 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 which 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 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(), which calculates whether the path created by the drone intersects any of perimeters of the no-fly zones, using the method linesIntersect() which calculates if two line segments intersect each other. The details for this are explained further below.

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 2d 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.

Moreover, 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 aid high cohesion within classes. As well as this, consider the line-line intersection problem for every line along the perimeters of multiple polygons resulted in several lines of code which was split into two methods, needing its own class to aid clarity of code and to prevent classes becoming too cumbersome.

Order

Restaurant

Section 2: Drone Control Algorithm

The Algorithm employed by the drone is a greedy approach.

Pre-Flightpath 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 flightpath generation

Once the orders have initially been sorted, then the algorithm iterates through all of the valid orders, in this sorted arrangement. For each order, the drone generates a part of the flightpath 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 flightpath on an individual move by move basis, adding the move to the partial flightpath before generating the next move. The algorithm calculates the next move by firstly 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, which involves checking 3 separate criteria: intersection with the no-fly zone, leaving the central area, and repetition of moves, each of which is detailed further 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 >> 2000 * 0.00015. If the move we are considering is an improvement on this distance we then select that move as a new best move. After all legal moves have been considered, we then put that move into the partial flightpath for the current order and consider the best move from the new position.

Backtracking

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 the original section of the flightpath to this point is reconstructed in reverse order, towards Appleton Tower. This process is done by iterating through all moves in the flightpath, in reverse order, then taking the angle of that move add 180 (modulo 360) and then calculating the position of the drone after this new move. We then add this reversed move to the partial flightpath. As this is backtracking the drone's flightpath, we find that the position of the drone after this new move is almost identical to the position of the drone on the path to the restaurant, except for some small rounding error due to floating point arithmetic, which is insignificant when the co-ordinates are changed from a double to a float when writing to outputted files.

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 have obtained a complete section of the flightpath for the specific order. We consider if adding this section to the current flightpath 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 flightpath and do not deliver the order, otherwise we add it to the full flightpath 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 firstly 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 the line segments formed between each adjacent pair of these vertices. This set is then the set of straight lines each along the perimeter of the polygon.

We then consider whether the line segment 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. 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 where along each line segment the intersection point of the two lines will be by:

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

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

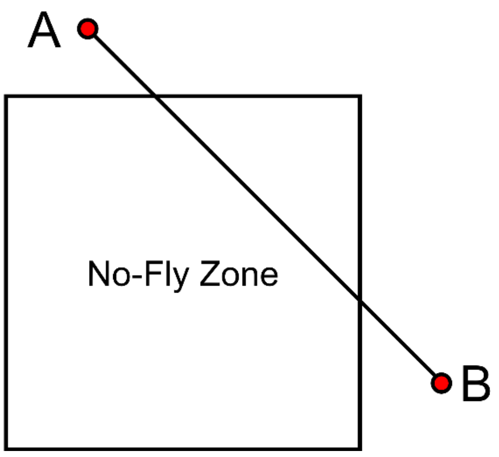

Figure 1: Bad approach - end points not in the No-Fly Zone, but the drone still goes through it

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.

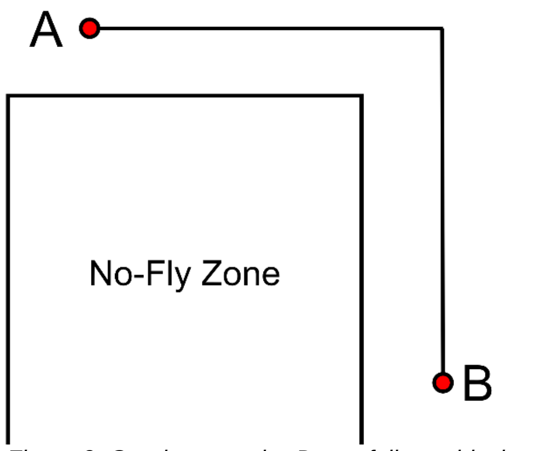

Figure 2: Good approach - Drone fully avoids the No-Fly Zone

This solution is calculated for each line segment formed by our polygon. If none of these line segments intersection then the drone move stays away form this part of the no-fly zone. We then check this for every polygon within the no-fly zone. If none of these polygons are intersected, the move is valid in terms of 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 end point 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: Bad approach - end points not in the No-Fly Zone, but the drone still goes through it. So we use the line-line intersection approach which, although less efficient, correctly identifies paths which fully avoid the no-fly zone, as shown in Figure 2: Good approach - Drone fully avoids the No-Fly Zone.

Staying within the central area

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 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.

Move repetition

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: Drone gets stuck when allowing repetition. 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, in comparison to point A. 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.

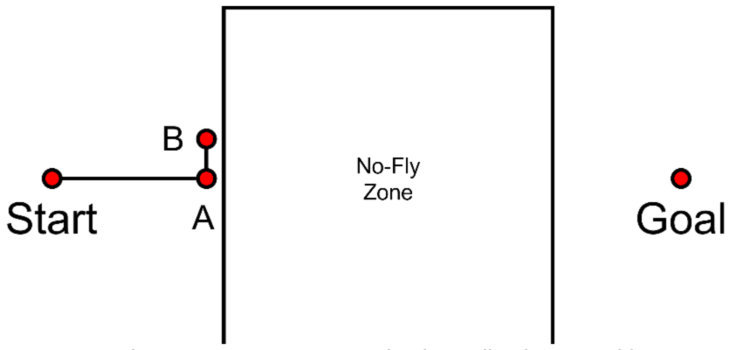

Figure 3: Drone gets stuck when allowing repetition

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, and only considering moves that would enter new points. So our flightpath 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 obviously allow positions to be repeated from the original flightpath to the restaurant, and observe that this section of the flightpath 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 made whilst attempting to pick-up the same order.

Improvements

One improvement that could be made to the algorithm to improve optimality of the drone's flightpath could be to consider the reversed approach to the flightpath algorithm as well. That is, firstly consider the flightpath that is created by the greedy move-by-move algorithm starting from the restaurant and travelling towards Appleton Tower and then backtrack the path from Appleton Tower to the restaurant. Then swap the first and second halves of the flightpath, to create a new flightpath that may be different to the one created by starting at Appleton Tower. Then the drone would take whichever of the two flightpaths is the shortest. This new approach taking the best of 2 paths would mean the result of the flightpath would be at worst the same as the original approach, and potentially better producing a shorter flightpath 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 flightpath 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 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 over 100 moves long the number of nodes that would need to be stored in memory was deemed infeasible as the memory requirement for A* is $O(b^d)$, 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* [1] might allow for a path to be created where there is a guarantee of optimality, with feasible memory requirements.

Figures

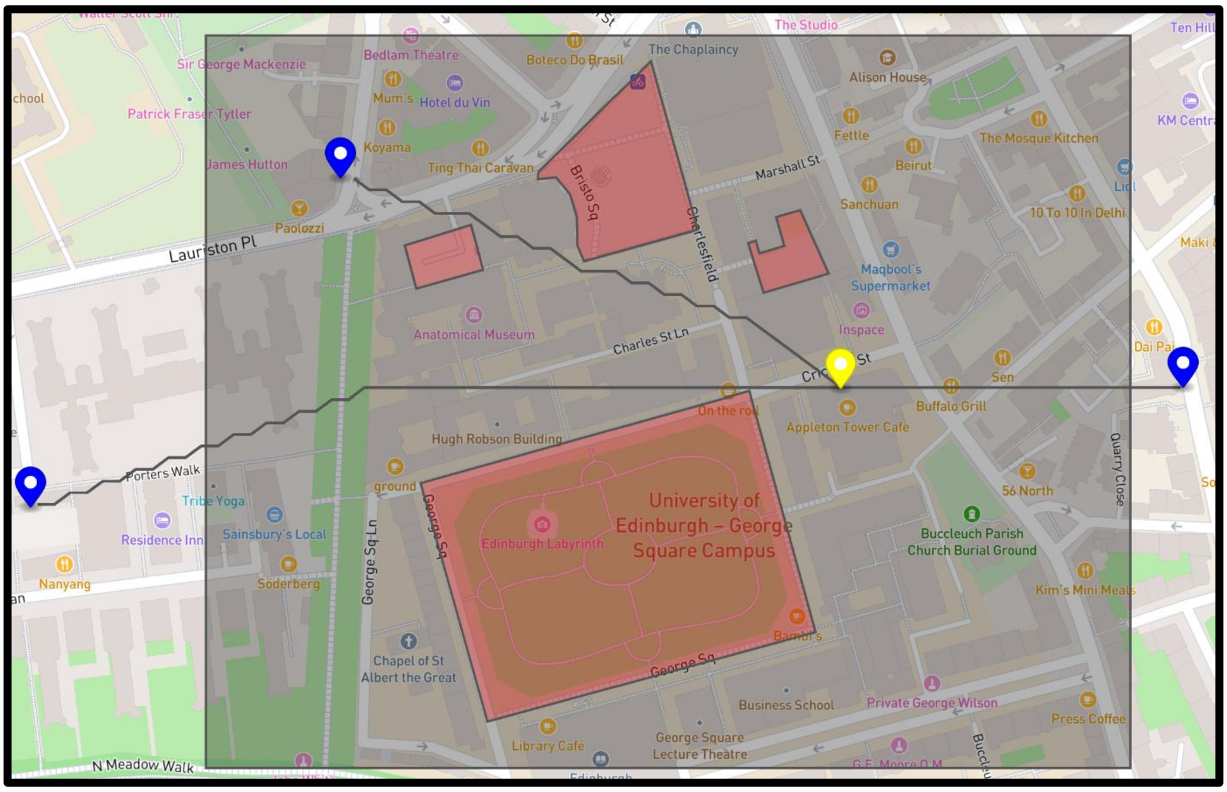

Figure 4 - Flightpath on 2023-02-19

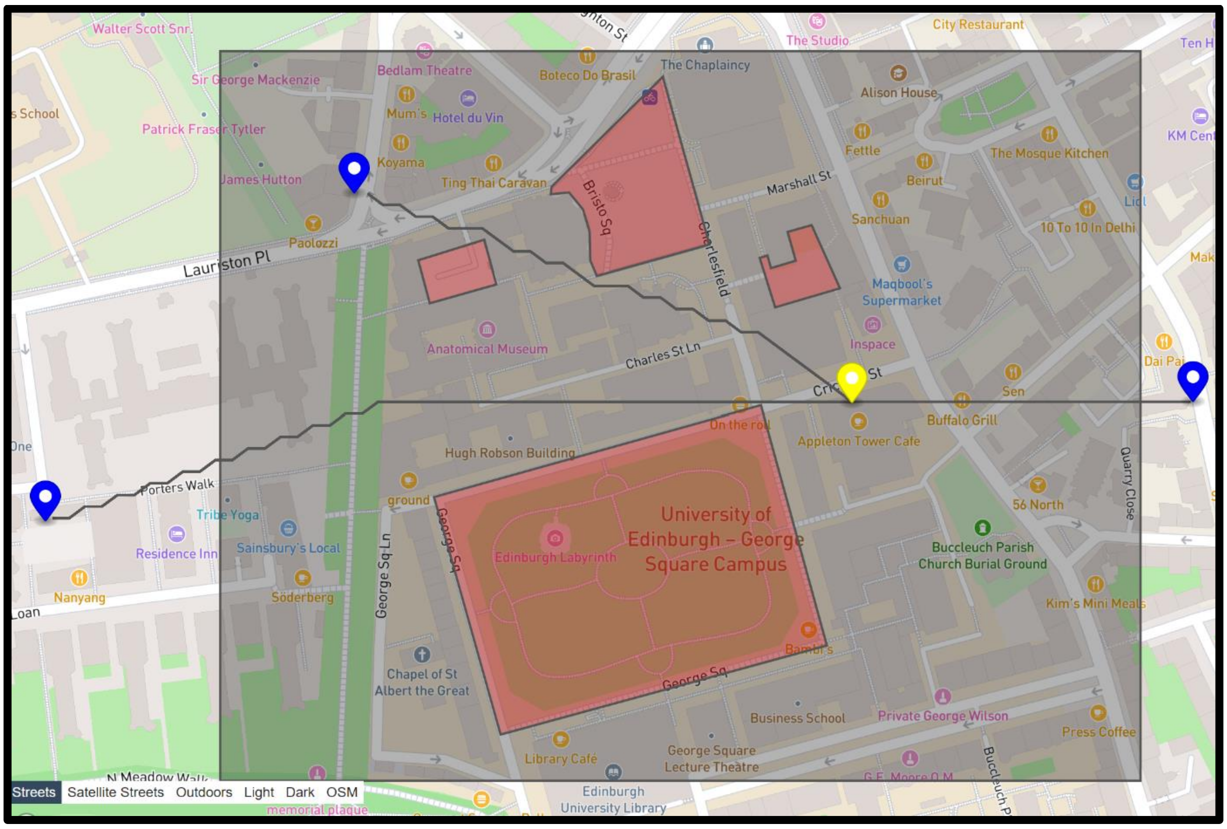

Figure 5 - Flightpath on 2023-04-15

References

[1] S. Russell and P. Norvig, "Artificial Intelligence: A Modern Approach," 2009, p. 101.

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

[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.