Informatics Large Practical

Coursework-2 Report

S1936373

This report contains the documentation for the ILP project that involved a complete implementation of the drone delivery service that delivers lunch orders placed at a given date.

Table of Contents

Software architecture description

The application I made makes use of a collection of Java classes that I identified as being the right ones for the application. These classes helped in the processing of command-line arguments, accessing tables from databases and contents from web servers at provided ports, controlling the drone movement taking into account several stipulations, and more.

Following are the classes I made use of in my application:

1. App

The App class is the main hub of my application. The command-line arguments (which include the date, the lunch orders are placed for, the web server port number, and the database server port number where the web server and database are running), are passed to this class. These arguments are then processed in this class. Processing these command-line arguments is essential as it serves as the backbone of accessing contents from the web server and the database correctly. This class is responsible for calling methods that are needed for the proper functionality of the application. To begin with, it gets the orders placed at the given date from the database, creates the flightpath table which provides a detailed record of every move made by the drone while making the day's lunch deliveries, and creates the deliveries table which has an entry for every lunch delivery which the drone makes. It also gets the no-fly zones from the web server that basically consists of buildings that the drone cannot fly over. The class then calls the main method to control the drone movement which then returns all the coordinates the drone moves to. Finally, it creates a geoJSON file which can be imported to geojson.io to visualize the drone movement.

2. Database

The Database class is responsible for all the functionalities that require accessing and modifying tables on the Apache Derby database. This class gets a connection to the database through the database port number that was provided as a command-line argument to the App class. It consists of methods that help in getting contents from the orders and orderDetails table that contains information on the lunch orders placed at the given date and the details of which menu items are in the order respectively. It consists of a method to sort a list of orders placed based on a greedy approach, i.e, based on the total price of an order. This class is also responsible for containing methods that help in creating the flightpath and deliveries table and it also helps in inserting the necessary data into these tables.

3. WebServer

The WebServer class is responsible for all the functionalities that require accessing contents from the web server running at the port number that was provided as a command-line argument

to the App class. It consists of a method that enables us to get the no-fly zones, over which the drone cannot fly, from the web server. It also consists of a method that enables parsing a JSON file that contains all the details of the shops that are available for delivery.
Every order accessed from the database contains a delivery location that is present as What3Words[1] encoding. This is a novel location addressing system that is made up of three words. The WebServer class consists of a method to get the coordinates from the What3Words, each word representing a directory in the words folder on the web server.
It also contains a method which helps in finding the location of the shop from which a food item has been ordered and its price.

4. Drone

The Drone class is the main class responsible for calculating a flightpath for the drone which delivers the lunch orders placed for the date passed as a command-line argument to the App class to its best abilities before it returns close to its initial starting position, that is, Appleton Tower. This class contains several methods for handling the drone movement so as to make sure that the drone follows all the stipulations placed on it. It has methods to check whether the drone is confined within the drone-confinement zone and whether the drone is flying over the no-fly zones. It also has a method that checks whether the drone has enough battery after making a move to come back to its initial starting position, that is, Appleton Tower.

5. Shops

The Shops class is responsible for representing the details of all the shops. It contains the name, location, and the menu containing all the food items available at the shop and the price of each item.

6. MenuItems

The MenuItems class is responsible for representing the details of an item. It contains the name and price of the item.

The Shops and MenuItems class are useful as they provide a way to parse the JSON file on the web server that contains all the information of the shops available to make delivery from.

[1] The What3Words system maps the earth's surface using 3m × 3m tiles, each with a unique three word address. More information is available at https://what3words.com/.

7. ItemPriceShopLocation

The ItemPriceShopLocation is responsible for representing the details of an item. It contains the item price and the location of the shop that contains the item on its menu.

8. Orders

The Orders class is responsible for representing the contents of the orders and order details table from the Apache Derby database. This class is essential as the Drone class makes use of contents from this class in order to calculate a flightpath for the drone. An object of the Orders class contains an order number, the delivery date, the customer student Id, the location to deliver to encoded as What3Words, the coordinate of the location to deliver to, the item ordered, the coordinates of the shop to get the item from and the item price.

9. Deliveries

The Deliveries class is responsible for representing an entry for every lunch delivery which the drone makes. It contains the order number of the order delivered, the location where the order was delivered encoded as What3Words, and the total cost of the order including the delivery fee. It also contains a method that provides the last order that is delivered which is useful when the Drone class is creating a list of orders delivered. This class is essential as the Drone class makes use of the Deliveries class when creating a list for every lunch order which the drone makes. The list of these deliveries made is then inserted into the deliveries table in the Apache Derby Database.

10. Flightpath

The Flightpath class is responsible for representing a detailed record of every move made by the drone while making the day's lunch deliveries. It contains the order number of the order the drone is currently delivering, the longitude and latitude at the start of the move, the angle of travel of the drone in the move, and the longitude and latitude at the end of the move. This class is essential as the Drone class creates a list of these Flightpath objects till the drone returns back to its initial location, Appleton Tower, which are then inserted into the flightpath table in the Apache Derby Database.

Drone control algorithm

Greedy Algorithm

My application makes use of a greedy approach. When it gets the list of orders placed, it first sorts the list based on the total price of an order. The order with the highest delivery cost for the items gets delivered first and the order with the lowest delivery cost for the items gets delivered last. Even though this approach isn't as effective as other methods like A* heuristic or Nearest insertion heuristic, it still has a high sampled average percentage monetary value[2].

Getting the Destination Coordinates

Once the list of orders placed is sorted, the algorithm loops over the list of orders and gets the destination coordinates which could either be the shop location where the item is available or the delivery location where the items need to be delivered. At the end of every iteration, there is a temporary variable that points to the current order such that when the next iteration starts, the temporary variable is still pointing to the previous order. Initially when the loop first starts, the destination coordinate is the shop location of the first order. For the next iterations of the loop, it checks whether the previous order number is the same as the current order number. If it is equal, then the destination coordinate is the shop location of the previous order. If it is not equal, it checks whether the destination coordinate is the previous shop location, if it is equal, then the destination coordinate is the delivery location of the previous order, and if it is not equal, then the destination coordinate is the shop location of the previous order. The loop takes into account the same order for the next iteration so that it can set the destination coordinate as the delivery location in order to deliver the order.

Moving the Drone

The class has a private field *moves* first initialized to 0 that holds the total number of moves made by the drone. Our algorithm also contains a *positions* list that first only contains the coordinates of Appleton Tower but as the drone moves, it keeps appending these positions into the list.
Once the algorithm gets a destination coordinate, it jumps into a while loop that checks whether the current position of the drone is close-to[3] the destination coordinate.
If it is close, the drone hovers[4] at the current position, updates the current position of the drone, appends the updated current position to the *positions* list and increments *moves* by 1 if the

---

[2] The sampled average percentage monetary value is calculated by taking a random sample of days (say 7, 12, 24, or 31 days) and computing the average of the percentage monetary value delivered on each of those days. The percentage monetary value delivered each day is calculated as the total monetary value of deliveries made divided by the total monetary value of orders placed.
[3] The distance between the 2 coordinates is less than 0.00015 degrees.
[4] The longitude and latitude of the position remains the same for a move.

drone moved at least once from its previous destination to the current destination which basically checks whether the drone gets multiple items from the same shop.
If it is not close, the algorithm repeatedly finds the angle between the current position of the drone and the destination coordinate, rounds off the angle to the nearest tens place (3.4 degrees becomes 0, 18.2 degrees becomes 20), finds the remainder of the rounded angle with 360 to get an angle between 0 and 350 (extremas inclusive), and then moves the current position of the drone by 0.00015 degrees towards the resulting angle and stores it in a temporary variable. The current position is updated to the next position which is appended to the *positions* list and *moves* is incremented by 1 only if there are enough moves (≤ 1500) to go from the next position that is stored in the temporary variable to Appleton tower where the drone has to return before it runs out of battery, i.e, moves ≤ 1500.
The algorithm calls a method to find the number of moves from the next position to Appleton which basically follows the same approach as going from the current position to destination coordinate while updating the current position till it reaches close to the destination coordinate, where the current position is the next position stored in the temporary variable and the destination coordinate is Appleton tower. It returns a list of coordinates the drone moves to, to reach close-to Appleton tower.
The size of this list incremented by 1 represents the number of moves to reach close-to Appleton tower and hover for a move. If the number of moves > 1500, the algorithm calls the same method that finds the positions of the drone to move from the current position to Appleton tower and appends these positions to the *positions* list.

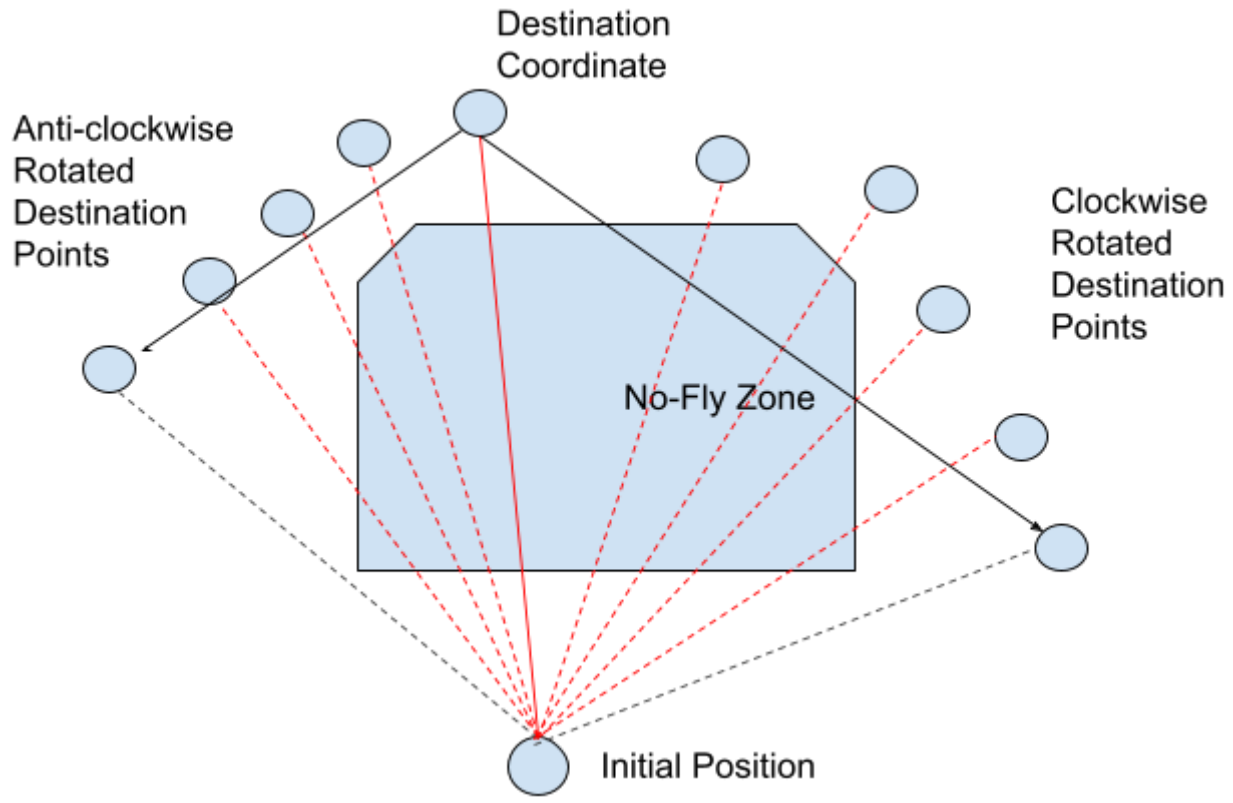
Problem with the movement of Drone

The problem with the movement is that there could be cases where when the drone moves, the position it moves to could either be outside the confined zone or in the no-fly zones.

Avoiding No-Fly Zones

In order to avoid no-fly zones, the algorithm makes use of the fundamental concept of checking whether 2 lines intersect.
In the previous while loop, if the current position of the drone is not close to the destination coordinate, after getting the resulting angle between the current position and the destination coordinate, the algorithm checks if the line between the current position and the destination coordinate intersect with any of the lines which make up the no-fly zones.
If they don't intersect, it checks whether the line from the current position to the next position(moving the drone from the current position by 0.00015 degrees towards the resulting angle) intersects with any of the no-fly zones. If the lines do intersect,(which is probable since we're rounding the angle between the 2 points), we check if adding 10(we pick 10 here as we're rounding the angle to the tens place) to the resulting angle leads to an intersection between current and next position. If it does, the new resulting angle becomes the resulting angle decremented by 10, and if it doesn't it becomes the resulting angle incremented by 10.
If the current position and destination coordinate intersect, the algorithm rotates the line made by the 2 around the current position in both clockwise and anticlockwise directions. The

algorithm keeps rotating the line in steps of 10 in an anticlockwise direction till the line does not intersect with any of the no-fly zones. Once it finds an angle, it rotates the destination coordinate around the current position making use of a transformation and rotation matrix, and gets the rotated destination point. It then finds the distance between the destination and rotated destination coordinate. If no such angle exists, distance is a number big enough to not be present within the confined zone.
Similarly, for clockwise direction, it gets the distance between the destination and rotated destination coordinate.
The algorithm picks the directional angle for which the distance between the destination and rotated destination coordinate is lower.
Once it gets the angle, we add the angle(positive for anticlockwise, negative for clockwise) to the resulting angle, call it rotation angle, and again check if the current position of the drone and the next position of the drone moved towards the rotation angle by 0.00015 degrees intersects with the no-fly zones. If it does we add 10 to the rotation angle and check again. If it intersects, the final rotation angle would be the rotation angle decremented by 10. If it doesn't, the final rotation angle would be the rotation angle incremented by 10.



The above diagram shows an example where the algorithm is used. Red lines symbolizing illegal moves. The algorithm will give 2 probable rotations, but it will pick the one where the line is rotated anticlockwise as the distance between this rotated point and the destination point is lower than the distance between the clockwise rotated point and the destination point.

Avoiding being outside Confinement Zone

In order for the drone to always stay within the confinement zone, every time the drone is about to move its position, the algorithm checks if the next position is confined to the confinement zone. The algorithm ignores the case where the next position of the drone after moving at some angle falls outside the confinement zone and iterates to the next possible angle.

My algorithm does not make use of the landmarks that are available on the web server as the algorithm I came up with worked better without using them.

Drone Flight Figures

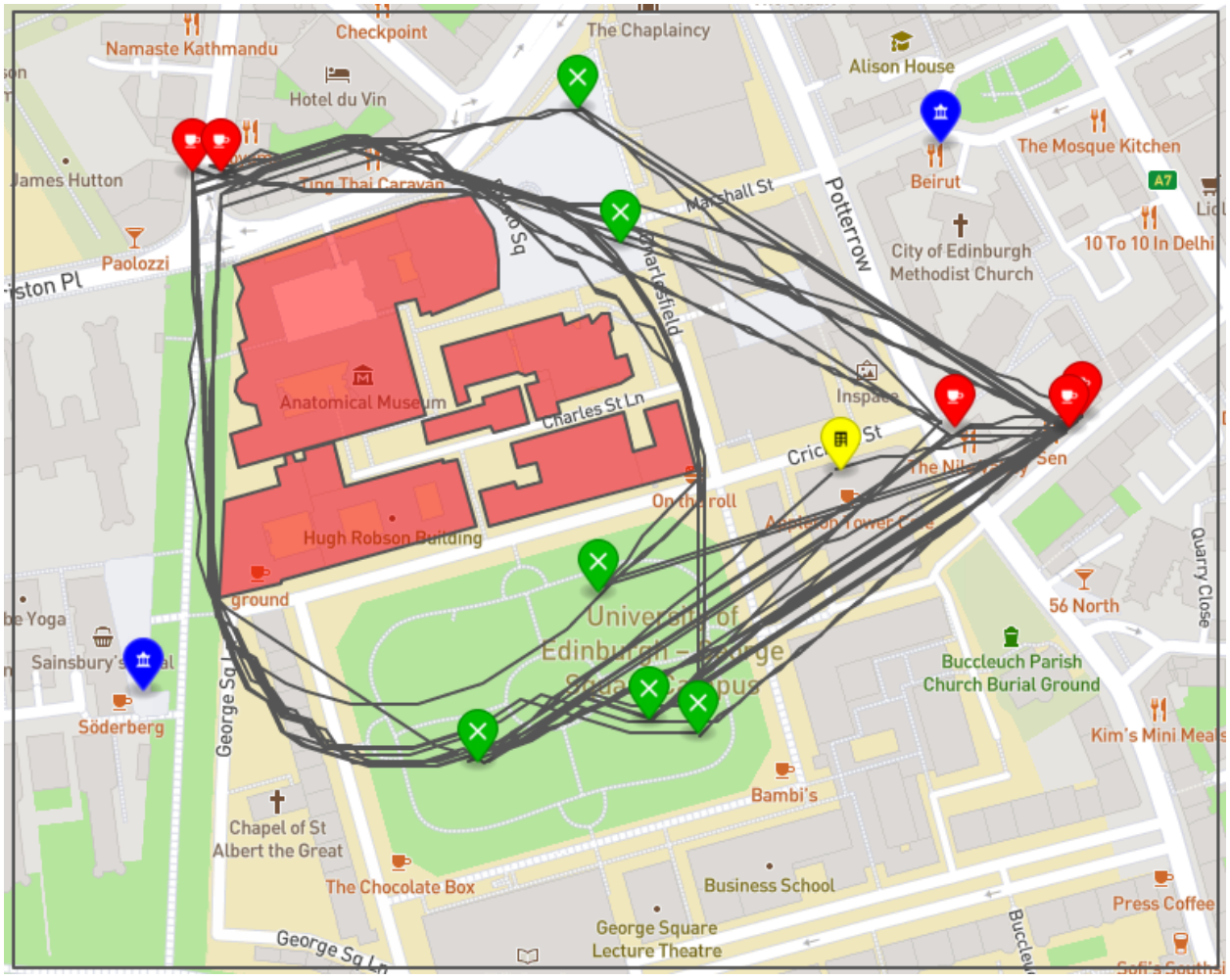


Figure 1: geoJSON output for deliveries made on 08/12/2023. The grey squiggly line connecting the initial location (the yellow marker), the shops (the red markers), the delivery locations (the green markers), and the final location (the initial location again) is a graphical representation of the flightpath of the drone which my algorithm generates

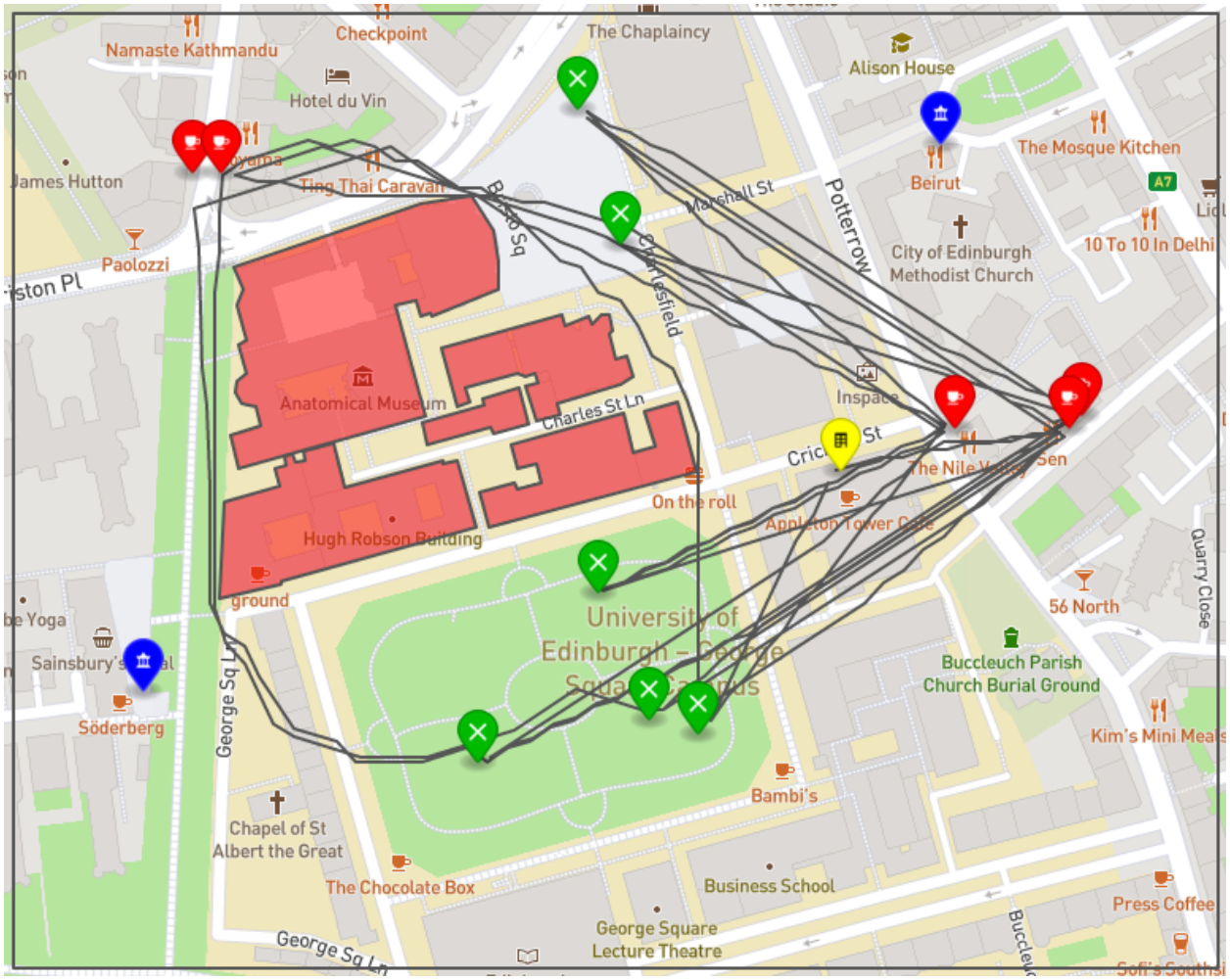


Figure 2: geoJSON output for deliveries made on 12/09/2022. The grey squiggly line connecting the initial location (the yellow marker), the shops (the red markers), the delivery locations (the green markers), and the final location (the initial location again) is a graphical representation of the flightpath of the drone which my algorithm generates