Informatics Large Practical
Coursework 2
Report

Maggie Lin
s1862667

Software Architecture Description

Outline

This section will provide a description of the software architecture of the implementation, and the importance of each.

Requirements

The goal of the project was to develop an implementation of a drone using Java which receives order details from a database and creates a flightpath to pick up and deliver these orders. This flightpath should avoid no fly zones as well as stay in the confinement area

Classes

My implementation consisted of 10 java classes, with a couple of inner classes. Primitive methods such as getters and setters will not be discussed. These classes are described in more detail below:

- **App:** This is the main class of the program, which handles the command line arguments. Using these arguments (specifically information on order dates and ports for the web server and database), this class instantiates objects of Database, Parser, Menus, Buildings classes. Methods to create the deliveries and flightpath table, retrieve orders for the day (from database) and the menu (from the web server) are called. These objects as well as orders and menu are used to instantiate an object of Drone class. A function to start the flightpath is called, as well as functions to read today's order details into the deliveries table. Lastly, a geojson map is created using the points travelled in the created flightpath and outputted to a new file.

- **Database:** This class contains all the functions involved in reading and writing from the Apache Derby database. This class is instantiated in the App class using the string to connect to "jdbc:derby://localhost:" + db + "/derbyDB" where db is the port where the database is running on. Within this function there are methods to create flightpath and deliveries tables which is where we output information generated from the startPath method in the Drone class.

Furthermore, there are two tables on the Apache Derby database, orders and orderDetails, which we use to receive order data for the day. We use the date to search the orders table for order information (such as orderNo, deliveryDate, customer, deliverTo), and then we use the order numbers found to find the items of each order. This data is parsed into an ArrayList of objects of Order class. Methods in this class are used to write information into the tables we created after the flightpath is created.

- **Order:** This class is used to represent an order. The orders' order number, delivery date, customer student number, delivery location and list of food items to be delivered is stored in this class. An ArrayList of this class is created using the findOrders method in Database class, and this represents all the orders of the day.

- **Parser:** This class contains all the functions which connect and receives information located on the web server. This class is instantiated in both App and Menu classes, using the port provided from the command line. There are three methods in this class. Using get requests and calling the send method on the client we created (in App class) two methods retrieve buildings details (either landmarks or no-fly-zones). These methods return a list of features and a list of menuDetails respectively. The other method is used to parse and split a What3Words string and connecting to the server to return a point which corresponds to the split 3 words.

- **W3W:** This class represents a What3Words location. This class matches the What3Words data exactly. Inner classes Southwest, Northeast and Coordinates are used to represent coordinates to the southwest, northeast, and the W3W location coordinates itself. The What3Words data is parsed from the web server into this class. The getters are used to retrieve the longitude and latitude.

- **Drone:** This class is used to handle all the functionality required to create the drone flightpath. Further detail will be provided in the drone control algorithm section. After instantiation in the main App class, the main method startPath() uses the helpers methods to update the local variables, list of flightpath points, angles, unvisited orders, and visited orders. The flightpath points are used to create a feature collection of a single linestring which is used in the App class to create the geojson map.

- **Moves:** This class is used to track the moves (angles and points travelled) for each order. It is specifically instantiated in the Drone class to track moves made per order, and to track moves needed to return to Appleton Tower at the end of the day.

- **LongLat:** This class is concerned with representing a point, such as the drone current position, as well as other useful locations. The methods in this class are concerned with the represented point's relationship with other points and angles as well as the confinement area. These functions include one to see if the point is within the confinement area, one that calculates the distance to another point, one that checks if a point is close to another, one that returns the next position that is travelled given an angle (moving at a length of 0.00015), and finally one that calculates the angle to another point. These are all used heavily in the drone algorithm and are converted to geojson points when the flightpath is created.

- **Menus:** This class has one method which is used to calculate the cost of a delivery in pence. It does this using a hash map of items to price. Iterating through the hash map and also iterating through the list of items provided in the Order class, if the items match, the price is added to the total. After this, the delivery cost of 50 pence is added to the total and returned. The Menus class also contains an inner class that matched JSON list exactly, so menus json data can be parsed directly into this class. This cost in pence value is used in App class to write to the deliveries table.

- **Buildings:** When this class is instantiated in the App class using the parser object, the list of landmarks and no fly zones are created as lists of points and polygons respectively. Using code reworked from the source: https://www.geeksforgeeks.org/check-if-two-given-line-segments-intersect/, I created three helper functions to determine if three points are collinear, to find the orientation of three points, and to determine if line segments intersect. Using these three helper functions, the isInNoFlyZone method was created to determine if a move between two points crosses a list of no-fly-zone polygons.

Third-Party Software

My implementation utilised these third-party dependencies (also shown in pom.xml file):

- JUnit – This is used to test some aspects of the implementation.
- Apache Derby Client JDBC Driver – This is used to connect to the database.
- Mapbox Java SQK GeoJSON – This is used to handle and make GeoJson features.
- Gson API – This is used to parse JSON files to be used.

Drone Control Algorithm

Requirements

As stated in the requirements the drone should:

- Make at most 1500 moves before it runs out of battery
- Cannot fly in an arbritary direction (only 0-350 in multiples of 10)
- Fly or hover
- When hovering, use angle -999
- Must hover for one move when picking up or delivering an order
- Start at Appleton Tower and end the day close to Appleton Tower
- Stay in confinement zone and stay out of no-fly-zone

Greedy Implementation

To implement a greedy pathfinding algorithm, I created a method which would find the best order to complete next, whilst there are still orders to visit. It does this by iterating through unvisited orders and calculates the score which is equal to price/distance. We want the highest value of this possible as these orders have the most value per distance travelled. There is a limitation in this method as I did not include having to move to landmarks, which would increase the distance, but I assumed this wouldn't the algorithm drastically.

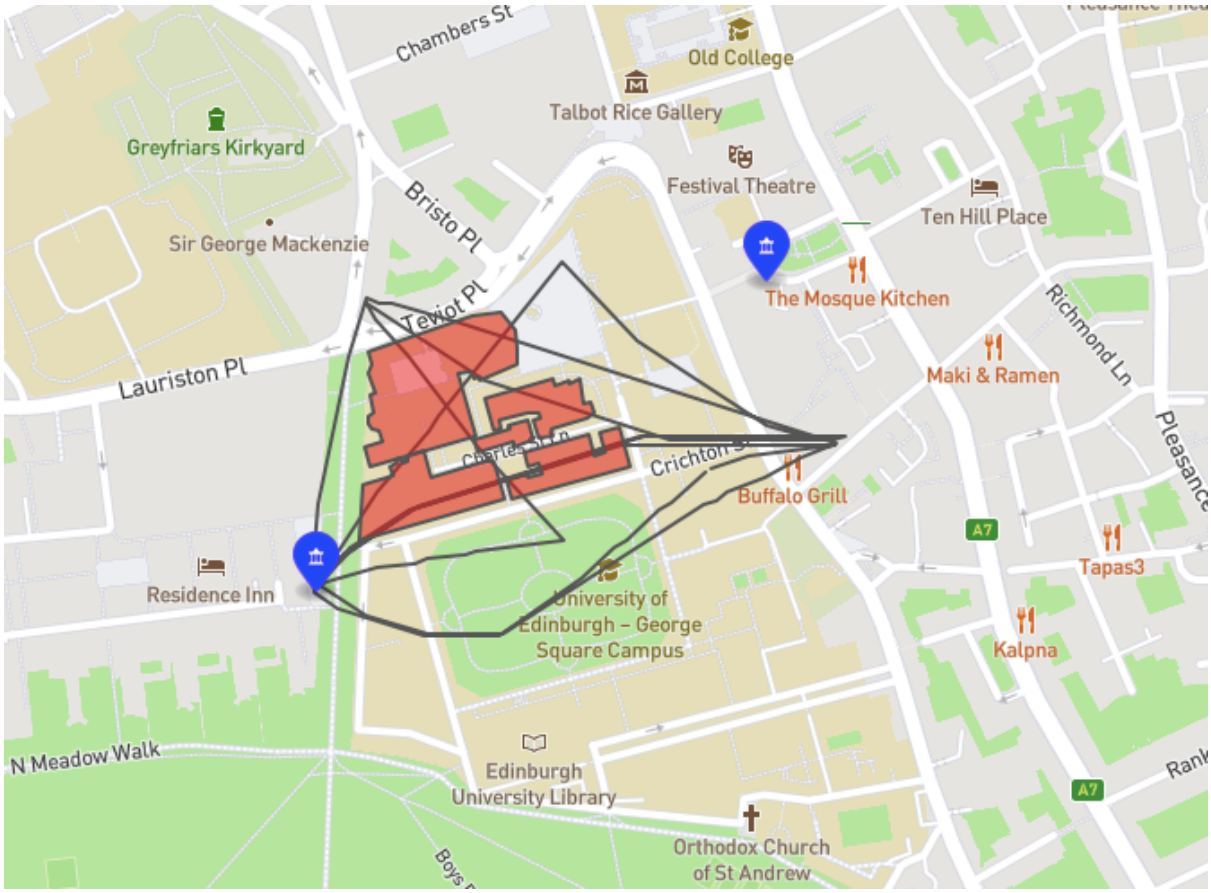
Staying Within Moves Limit

To make sure there would be enough moves to go back to Appleton Tower at the end of the day, for each order, I instantiated a Moves class and used the size of the number of angles to track how many moves the order would take. I also kept track of the original LongLat position in case we had to go back (if there are not enough moves). After using the moveTo method to pickup and deliver the order, we performed a check (totalMoves + moves.angles.size() + movesToAT(currentPosition) > maxMoves) and if this was true, the order would not be visited and the position would go back to the original position. Otherwise, the order would be confirmed, and the flightpath angles and points would be updated, as well as the number of total moves. The flightpath information is furthermore written to the flightpath table in the database. This is the control flow which I used to make sure the total number of moves made did not go over 1500, so the drone does not run out of battery.

Staying in Confinement Zone

To make sure all the points travelled by the drone stayed in the confinement area, I just used the isConfined method in LongLat to make sure they were all confined afer each move.

Avoiding No-Fly-Zone

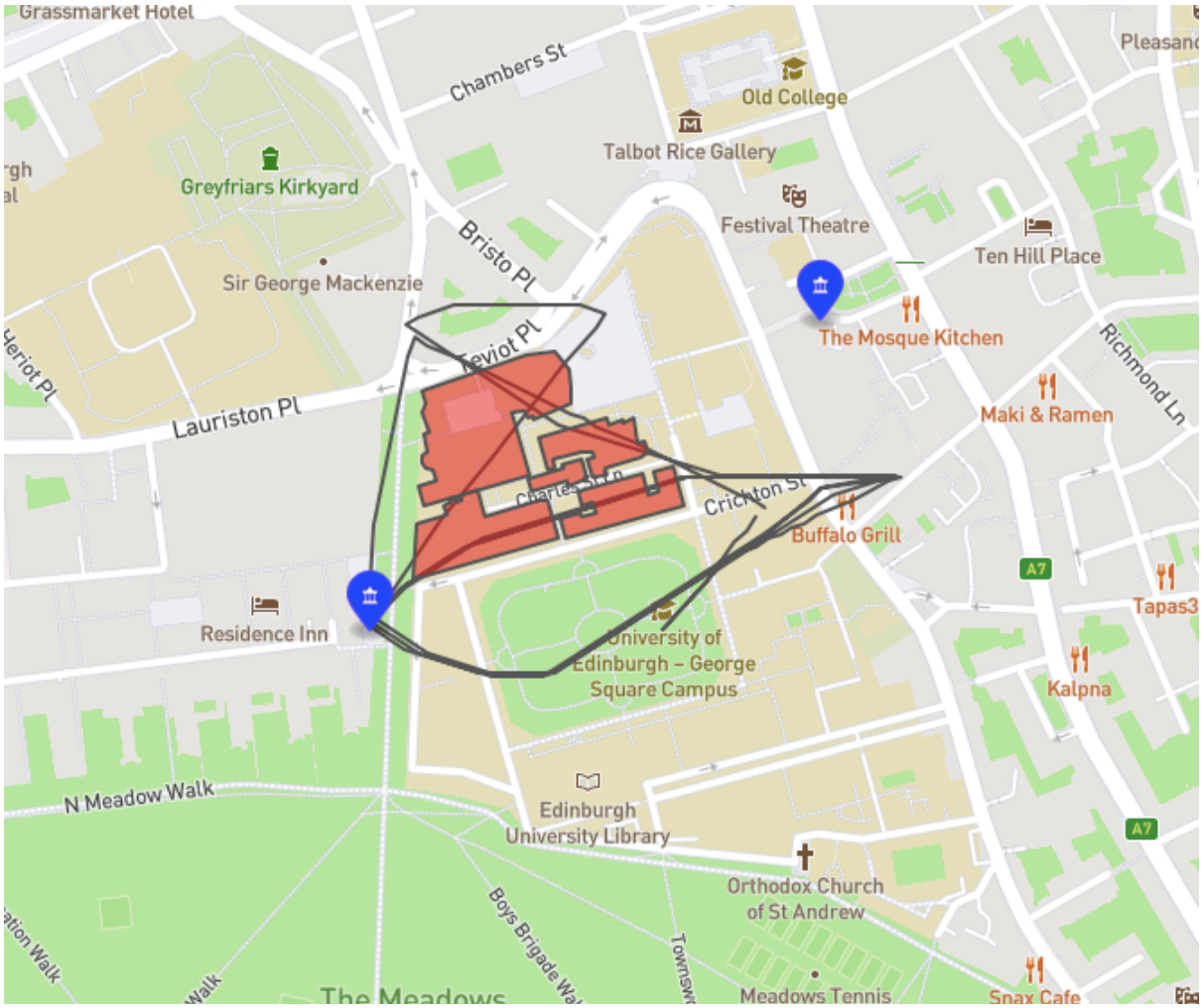


Map created on 01/01/2022

This is the part where I struggled in this implementation.

In my buildings class, like I described above I tried to solve the problem using line-line intersection (line between points and line of perimeter of polygon). However, the line was still going through the no-fly-zone boundary.

Furthermore, I decided to use closest landmark to fly to when a move does go through a no-fly-zone which resulted in the left landmark being used for all the maps, which is not ideal.



Map created on 02/02/2022