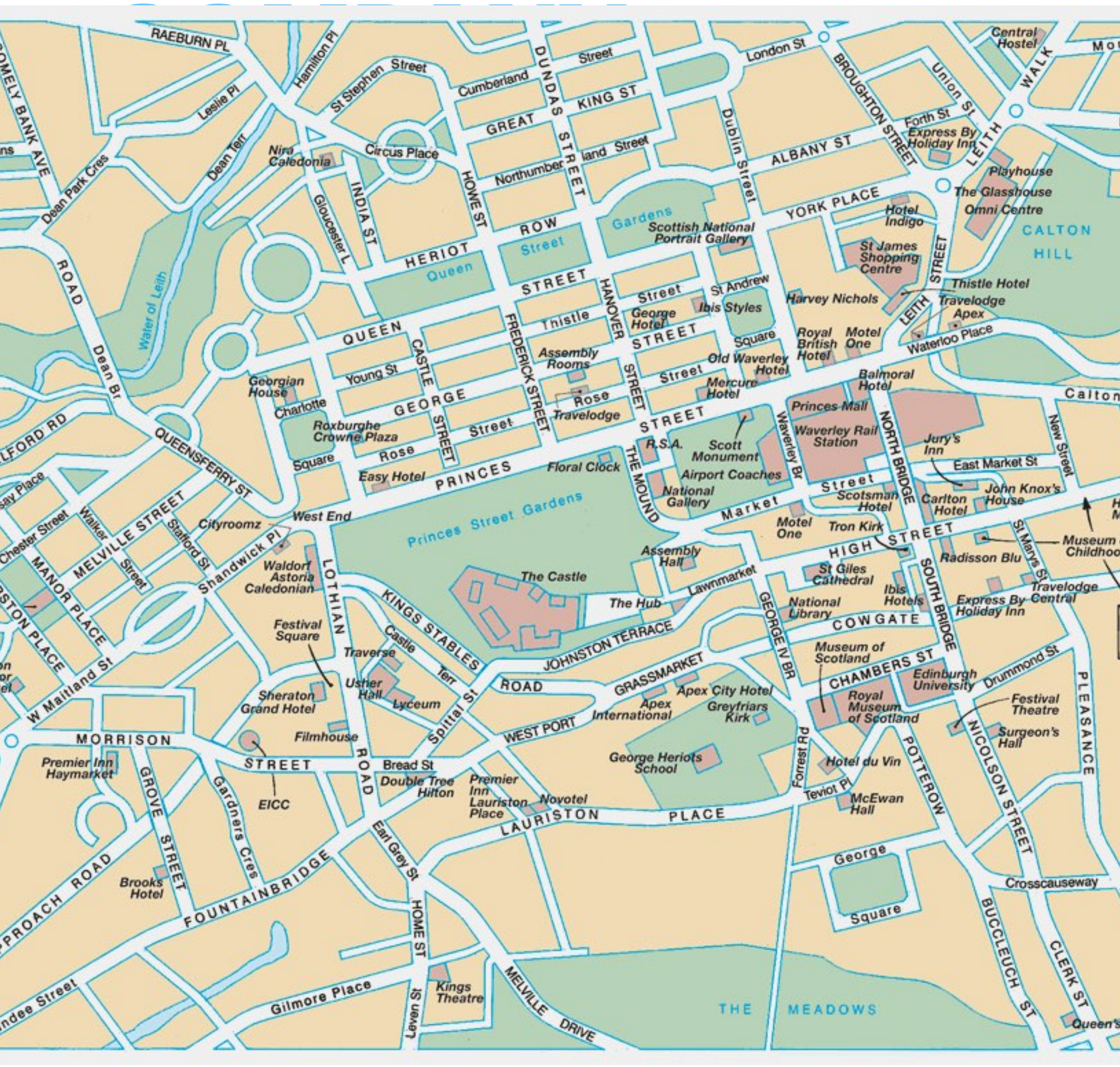
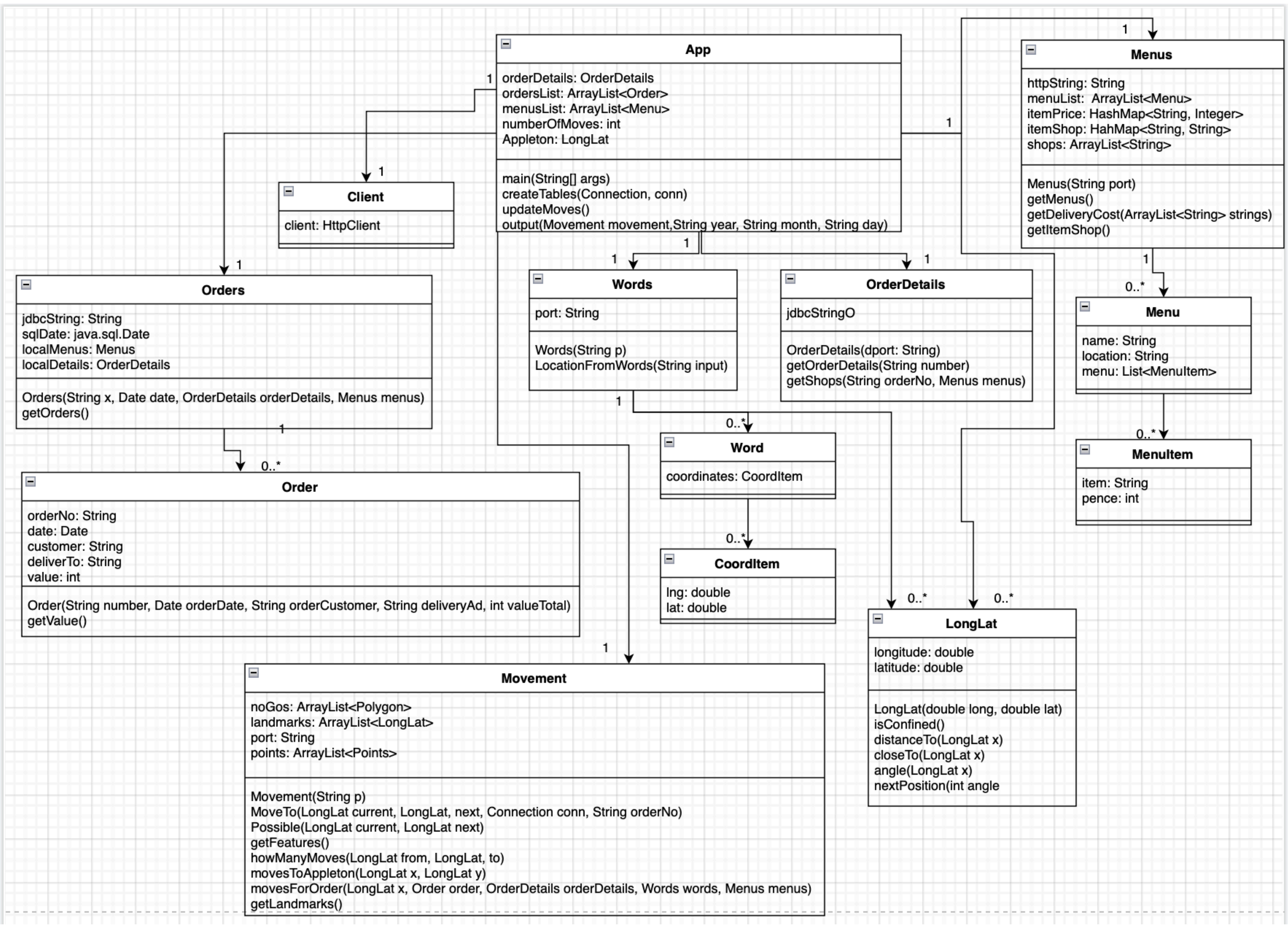
INFORMATICS LARGE PRACTICAL REPORT



EOIN REID, S1858933.

SOFTWARE ARCHITECTURE DESCRIPTION

Below is a UML diagram representing my project. It shows each class alongside important variables, methods and relationships.


App
orderDetails: OrderDetails
ordersList: ArrayList<Order>
menusList: ArrayList<Menu>
numberOfMoves: int
Appleton: LongLat
main(String[] args)
createTables(Connection, conn)
updateMoves()
output(Movement movement,String year, String month, String day)
Menus
httpString: String
menuList: ArrayList<Menu>
itemPrice: HashMap<String, Integer>
itemShop: HahMap<String, String>
shops: ArrayList<String>
Menus(String port)
getMenus()
getDeliveryCost(ArrayList<String> strings)
getItemShop()
Client
client: HttpClient
Orders
jdbcString: String
sqlDate: java.sql.Date
localMenus: Menus
localDetails: OrderDetails
Orders(String x, Date date, OrderDetails orderDetails, Menus menus)
getOrders()
Words
port: String
Words(String p)
LocationFromWords(String input)
OrderDetails
jdbcStringO
OrderDetails(dport: String)
getOrderDetails(String number)
getShops(String orderNo, Menus menus)
Menu
name: String
location: String
menu: List<MenuItem>
MenuItem
item: String
pence: int
Word
coordinates: CoordItem
CoordItem
lng: double
lat: double
Order
orderNo: String
date: Date
customer: String
deliverTo: String
value: int
Order(String number, Date orderDate, String orderCustomer, String deliveryAd, int valueTotal)
getValue()
LongLat
longitude: double
latitude: double
LongLat(double long, double lat)
isConfined()
distanceTo(LongLat x)
closeTo(LongLat x)
angle(LongLat x)
nextPosition(int angle
Movement
noGos: ArrayList<Polygon>
landmarks: ArrayList<LongLat>
port: String
points: ArrayList<Points>
Movement(String p)
MoveTo(LongLat current, LongLat, next, Connection conn, String orderNo)
Possible(LongLat current, LongLat next)
getFeatures()
howManyMoves(LongLat from, LongLat, to)
movesToAppleton(LongLat x, LongLat y)
movesForOrder(LongLat x, Order order, OrderDetails orderDetails, Words words, Menus menus)
getLandmarks()
1
0..*

CLASSES

APP: In this class, in the main method, we receive the command line arguments process them and then use them to create instances of the other classes in the project. We call methods from the other classes that read in all the data we need both from the web server and the database. Once we have instantiated the classes and read in all relevant data we create the deliveries and flightpath tables and begin to process our orders for the given day, completing our orders list order by order under the condition that we have enough moves left to do so. Once an order is complete we populate the deliveries table with the details for that order. The app class also contains the methods to create the database tables in createTables (called from main) to do so it first checks to see if tables under those names already exist, in which case it deletes them and creates new ones. We also have the method to update the number of moves we have remaining updateMoves which simply subtracts one from our total when called. Lastly we have the output method which creates a geojson file containing the line string showing our drones movements.

WORDS: The words class contains the locationFromWords method which is called from main and is used to get the location from a what3words string. To do so it accesses the relevant folder on the web server and reads the coordinates from this, before returning the coordinates as a LongLat object.

ORDERS: The orders class contains the getOrders method which is called from main and is used to obtain a list of Order objects, containing the orders for the specified day. In order to do so it queries the database for all order entries where the deliveryDate is the date specified at the command line at runtime. It then reads the data returned and creates Order objects adding these to a list before returning the list.

MOVEMENT: The movement class controls the movement of our drone, and implements the restrictions set out in the task specification. The moveTo method allows us to move from one LongLat location to another. It tests whether its possible to move directly to the desired location from the start location and if it is calculates the angle to the next location and moves step by step towards it until it is considered 'close to' (within 0.00015 degrees of it). After each move it populates the flightpath table with the details for that move including start location, end location and order number. If it is not possible to move directly to the desired location then the algorithm first moves to a landmark and then to the desired location by checking if such a move is possible first and then recursively calling moveTo to the landmark and then to the end location. Within movement we also have the possible method, used to check if it is possible to move directly to a desired location or if we have to move to a landmark first returning True and False respectively. We also have the getFeatures and getLandmarks methods which retrieve the no-fly zones and the landmarks respectively from the web server. Lastly we have the howManyMoves, movesToAppleton and movesForOrder methods. Each of these is designed to calculate the number of moves required to perform a task so that we can check we have enough moves left to do so.

ORDERDETAILS: The orderDetails class is used to obtain the details for a specified order. The getOrderDetails when given a class will return and array list containing the string values of each item in the order. It does so by querying the orderDetails database using the String representing an order number which it is given. It also contains the getShops method which when given an order number returns a list of what3words strings representing the addresses of the shops that order will visit.

MENUS: The menus class is used to get the menus for each shop. It contains the getMenus method called in App.java main which gets the menus for each shop and returns them in an array list of menu items. It does so by accessing the web server and reading the data from it. It also updates the itemPrice and itemShop hash maps allowing quick and easy access to shop location values and price values for any item in any order.

LONGLAT: The longlat class is used to create longlat objects and gives various methods that can be performed on them. It contains the distanceTo method which calculates the distance between two longlat objects, the closeTo method which returns true if a longlat is deemed to be close to another longlat and false otherwise, the angle method which calculates the angle between two longlats and returns it rounded to the nearest 10 and lastly the nextPosition method which, given an angle, returns a new longlat moved 0.00015 degrees in that direction.

RELATIONS

APP - WORDS, the app class has a one to one relationship with Words; there is only one instance of the words class needed per run of app

APP - ORDERS, the app class has a one to one relationship with Orders; there is only one instance of the orders class needed per run of app

APP - MOVEMENT, , the app class has a one to one relationship with Movement; there is only one instance of the movement class needed per run of app

APP - ORDERDETAILS, , the app class has a one to one relationship with OrderDetails; there is only one instance of the OrderDetails class needed per run of app.

APP - MENUS, the app class has a one to one relationship with Menus; there is only one instance of the menus class needed per run of app.

APP - LONGLAT, the app class has a one to many relationship with LongLat, many LongLat objects are created in the running of app.

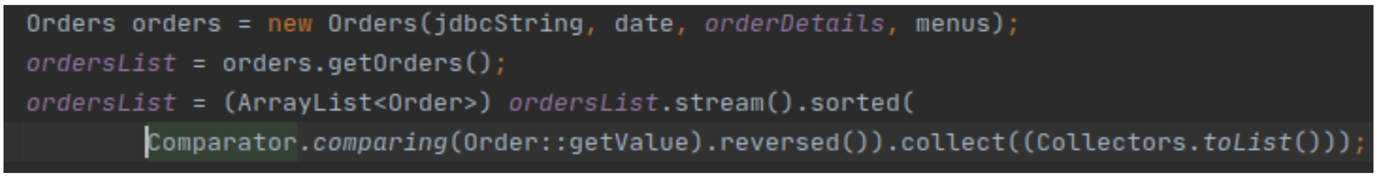
WORDS - LONGLAT, each time locationFromWords is run in the Words class one instance of LongLat is created so there is a one to one relationship between words and longlat.

MOVEMENT - LONGLAT, the movement class' methods create multiple LongLat objects and so there is a one to many relationship between movement and longlat.

ALGORITHM

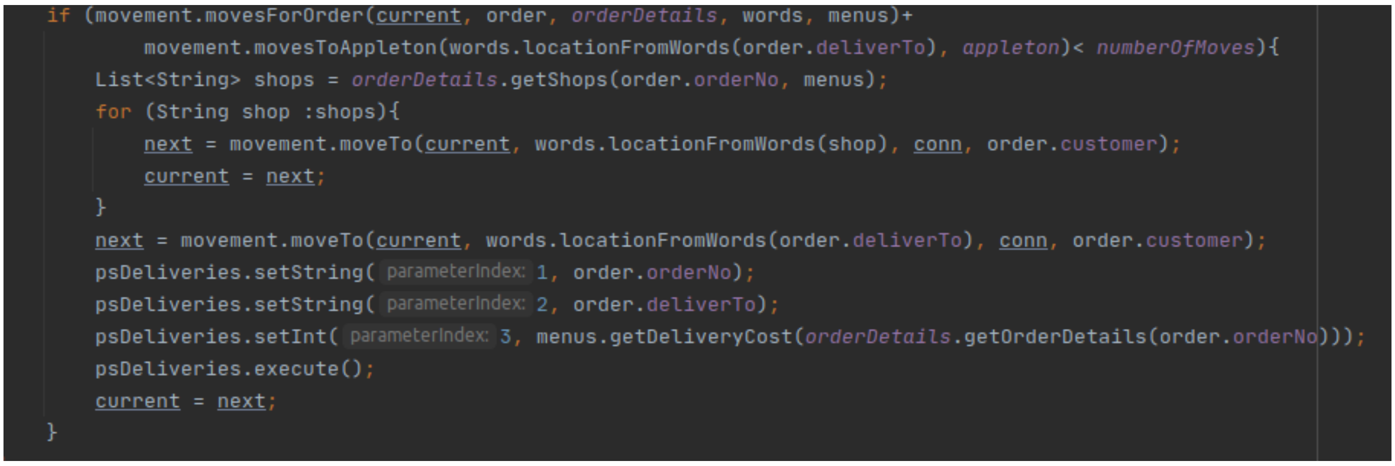
My algorithm to control my drone on its flight is as follows:

Once my program has the list of orders for the given day, it sorts the orders based on their total value, this means that when I go to deliver each order I deliver the most valuable orders first. This is my way of maximising the drone's score on the sampled average percentage monetary value. This is a greedy way of maximising the monetary value but much quicker than other methods I tried, giving a good trade off between value and runtime.



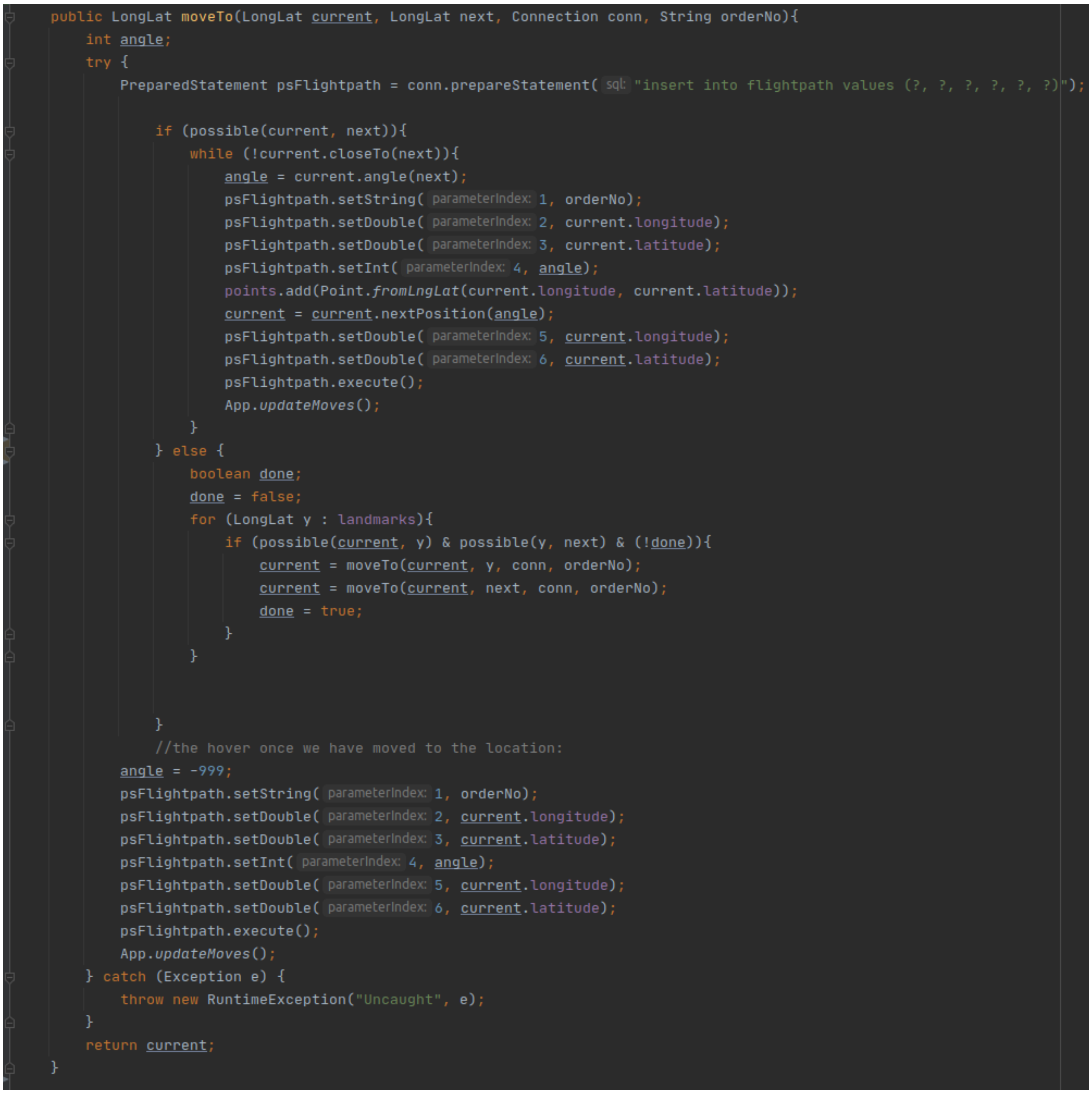
```
Orders orders = new Orders(jdbcString, date, orderDetails, menus);
ordersList = orders.getOrders();
ordersList = (ArrayList<Order>) ordersList.stream().sorted(
        Comparator.comparing(Order::getValue).reversed()).collect((Collectors.toList()));
```

The algorithm then takes each order one by one, first it checks how many moves would be required to complete the order and then return to Appleton. If this number of moves is greater than the number of moves we have left then we move on to the next order and try that. If it is less then first we complete the movements to each shop. By calling moveTo on the current location and each shop in the order (shops are obtained from getShops method).



```
if (movement.movesForOrder(current, order, orderDetails, words, menus)+
        movement.movesToAppleton(words.locationFromWords(order.deliverTo), appleton)< numberOfMoves){
    List<String> shops = orderDetails.getShops(order.orderNo, menus);
    for (String shop :shops){
        next = movement.moveTo(current, words.locationFromWords(shop), conn, order.customer);
        current = next;
    }
    next = movement.moveTo(current, words.locationFromWords(order.deliverTo), conn, order.customer);
    psDeliveries.setString( parameterIndex: 1, order.orderNo);
    psDeliveries.setString( parameterIndex: 2, order.deliverTo);
    psDeliveries.setInt( parameterIndex: 3, menus.getDeliveryCost(orderDetails.getOrderDetails(order.orderNo)));
    psDeliveries.execute();
    current = next;
}
```

The moveTo function will check if its possible to move directly to the given shop and if it is it will simply calculate the angle between the locations and apply nextPosition to the starting location until it is closeTo the given shop. If it is not possible to move directly to it then the algorithm will check if it is possible to go to one of the landmarks and then to the shop. For whichever landmark is possible it will then call moveTo to move to the landmark and then moveTo to move to the shop.

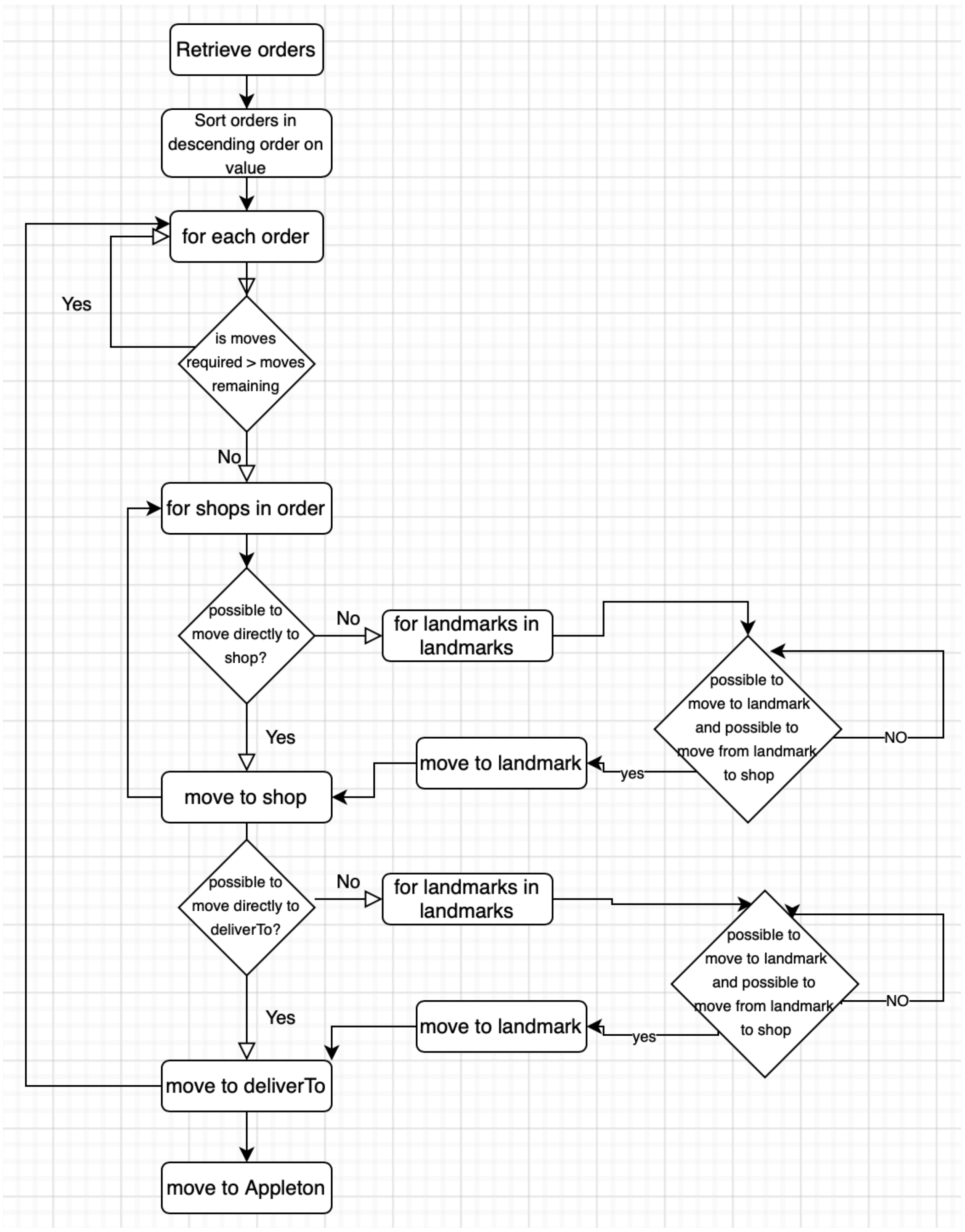


```java
public LongLat moveTo(LongLat current, LongLat next, Connection conn, String orderNo){
    int angle;
    try {
        PreparedStatement psFlightpath = conn.prepareStatement( sql: "insert into flightpath values (?, ?, ?, ?, ?, ?)");

            if (possible(current, next)){
                while (!current.closeTo(next)){
                    angle = current.angle(next);
                    psFlightpath.setString( parameterIndex: 1, orderNo);
                    psFlightpath.setDouble( parameterIndex: 2, current.longitude);
                    psFlightpath.setDouble( parameterIndex: 3, current.latitude);
                    psFlightpath.setInt( parameterIndex: 4, angle);
                    points.add(Point.fromLngLat(current.longitude, current.latitude));
                    current = current.nextPosition(angle);
                    psFlightpath.setDouble( parameterIndex: 5, current.longitude);
                    psFlightpath.setDouble( parameterIndex: 6, current.latitude);
                    psFlightpath.execute();
                    App.updateMoves();
                }
            } else {
                boolean done;
                done = false;
                for (LongLat y : landmarks){
                    if (possible(current, y) & possible(y, next) & (!done)){
                        current = moveTo(current, y, conn, orderNo);
                        current = moveTo(current, next, conn, orderNo);
                        done = true;
                    }
                }


            }
            //the hover once we have moved to the location:
        angle = -999;
        psFlightpath.setString( parameterIndex: 1, orderNo);
        psFlightpath.setDouble( parameterIndex: 2, current.longitude);
        psFlightpath.setDouble( parameterIndex: 3, current.latitude);
        psFlightpath.setInt( parameterIndex: 4, angle);
        psFlightpath.setDouble( parameterIndex: 5, current.longitude);
        psFlightpath.setDouble( parameterIndex: 6, current.latitude);
        psFlightpath.execute();
        App.updateMoves();
    } catch (Exception e) {
        throw new RuntimeException("Uncaught", e);
    }
    return current;
}
```

Once it has completed the moveTo sequence for each shop in the order it then moves to the deliverTO location for the order, calling moveTo on the last position the drone was at and the deliverTo location. Seen above after the for loop for shops.

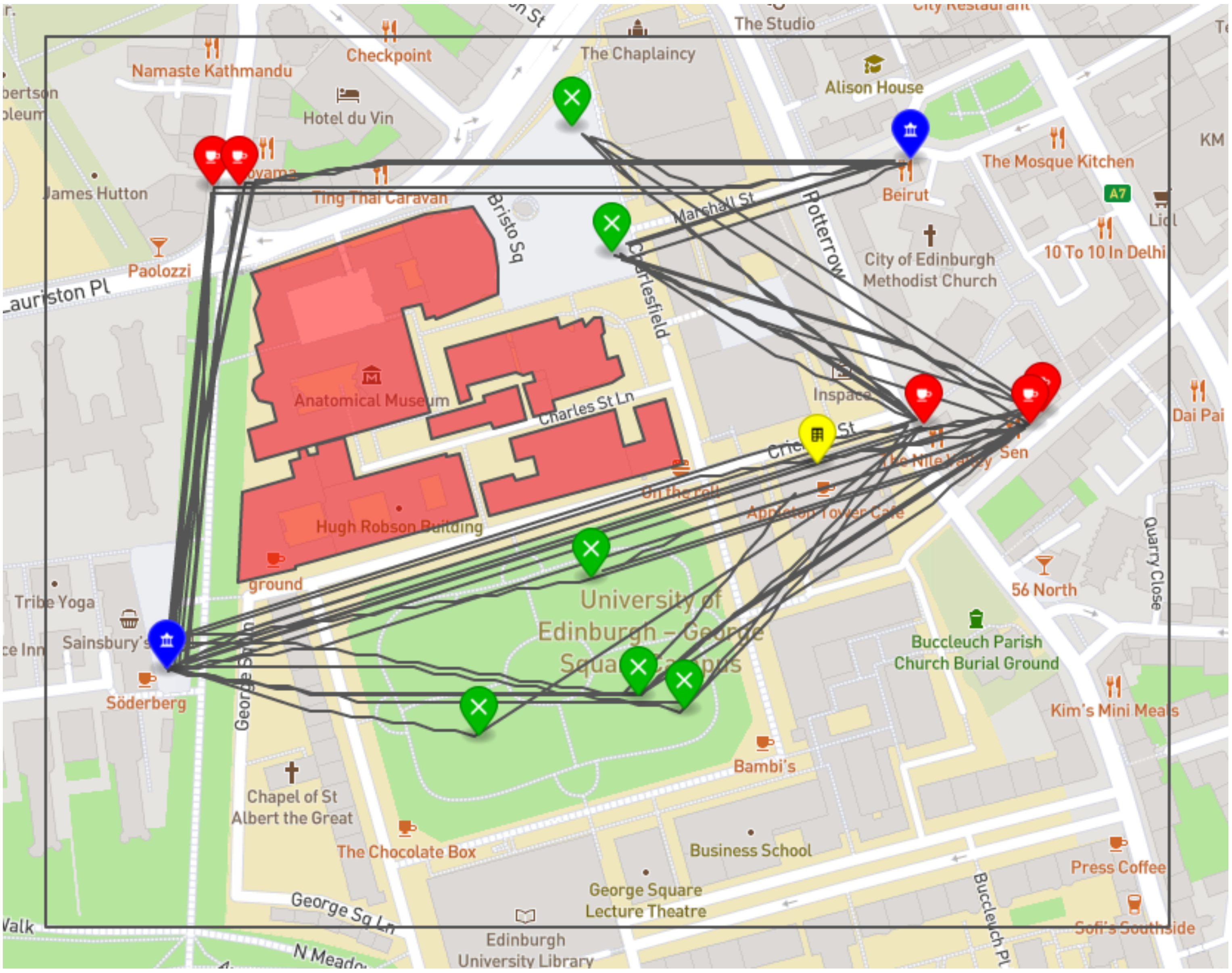
Finally once this process has been executed for every order on that day, executing the orders which the number of moves permits, the algorithm calls moveTo on the last location the drone was at and the address of Appleton, taking the drone back home after the days deliveries.

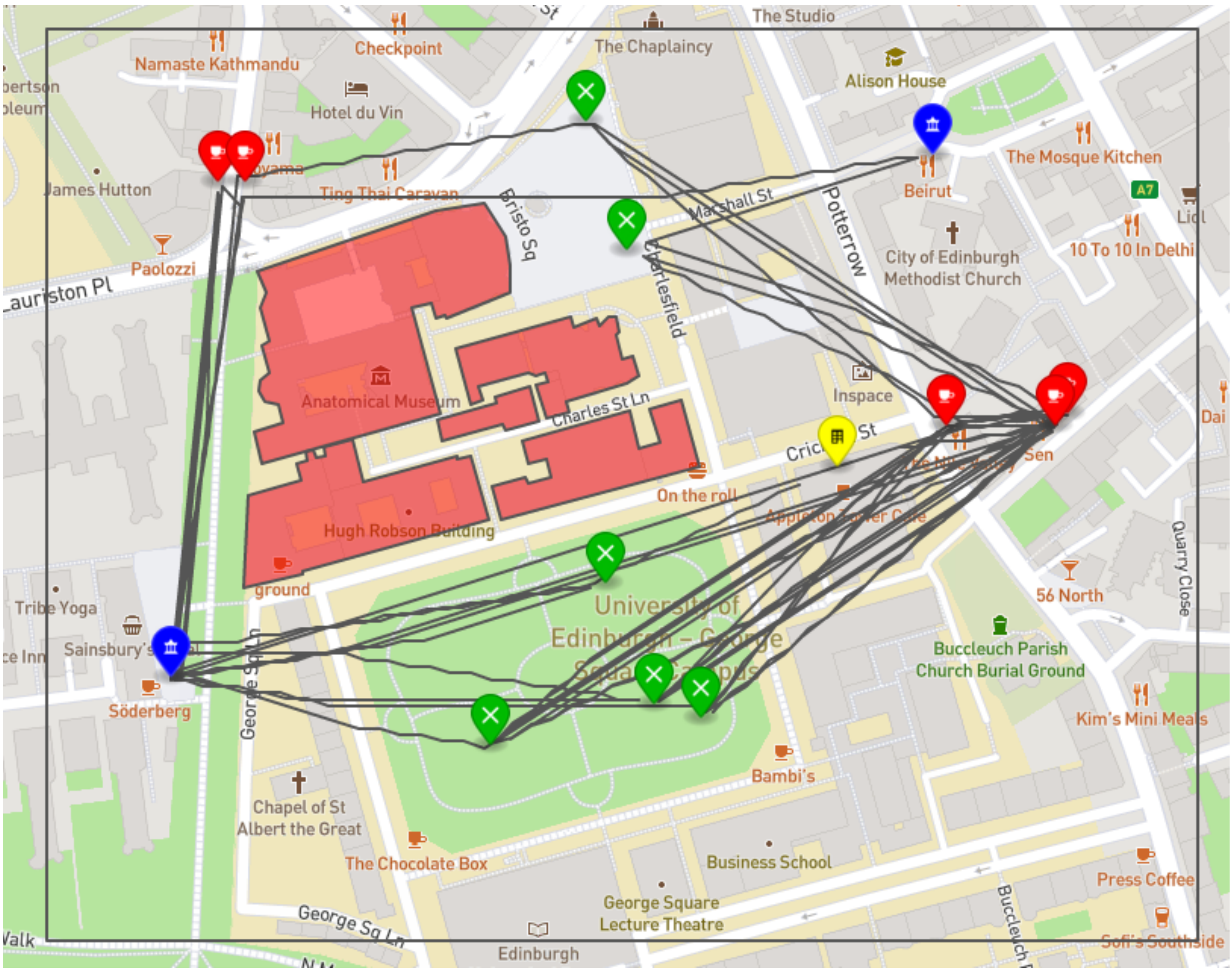
A flowchart for the algorithm is shown below showing the key processes (and omitting some less structurally important ones):



Retrieve orders
Sort orders in descending order on value
for each order
Yes
is moves required > moves remaining
No
for shops in order
possible to move directly to shop?
No
for landmarks in landmarks
possible to move to landmark and possible to move from landmark to shop
NO
yes
move to landmark
Yes
move to shop
possible to move directly to deliverTo?
No
for landmarks in landmarks
possible to move to landmark and possible to move from landmark to shop
NO
yes
move to landmark
Yes
move to deliverTo
move to Appleton

The output of two example runnings of the algorithm are shown below with the movements of the drone plotted in the grey line. The first shows the drones movements on the date 25-09-2023 and the second on the date 10-04-2023.





REFERENCES

https://stackoverflow.com

https://docs.mapbox.com/android/java/guides/geojson/

https://db.apache.org/derby/integrate/plugin_help/derby_app.html

https://www.codejava.net/java-se/jdbc/connect-to-apache-derby-java-db-via-jdbc

https://maven.apache.org/guides/introduction/introduction-to-the-pom.html