ILP Report

1.Software architecture description

1.1 Introduction

I have been tasked by the University of Edinburgh to implement an algorithm to plan the flightpath of a drone as it picks up orders from restaurants and delivers the orders to students at Appleton tower. The drone should deliver as many orders as possible before running out of moves (2000) while respecting the several constraints that the movement of the drone should follow. The purpose of this section is to provide an overall description of the software architecture of this application. I will explain why I think each java class worked well for the application. And the scope is that it provides an architectural overview of this project.

1.2 Non-Functional Requirements

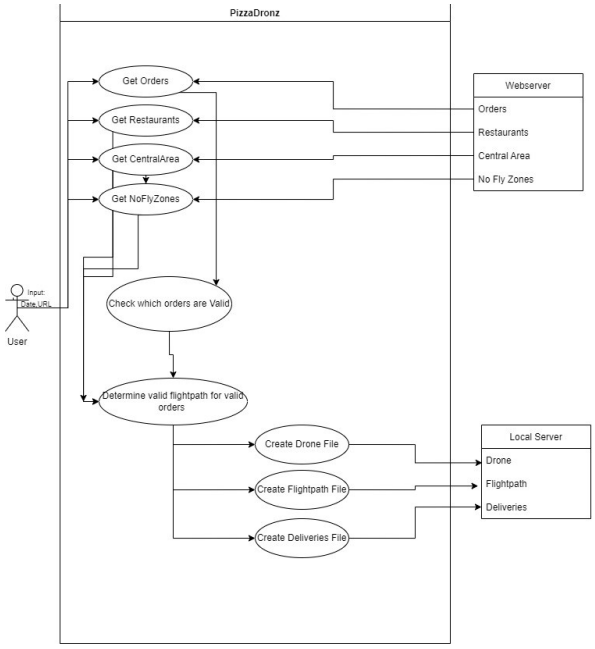
The non-functional requirements for this project include:

1. Performance – The planning and plotting of the drones flightpath should aim to have a runtime of 60 seconds or less.
2. Secure – functions and variables should be private where possible.
3. Robustness – The application should be able to handle unexpected inputs, exiting the program when necessary and should avoid any run-time errors.

1.3 Constraints

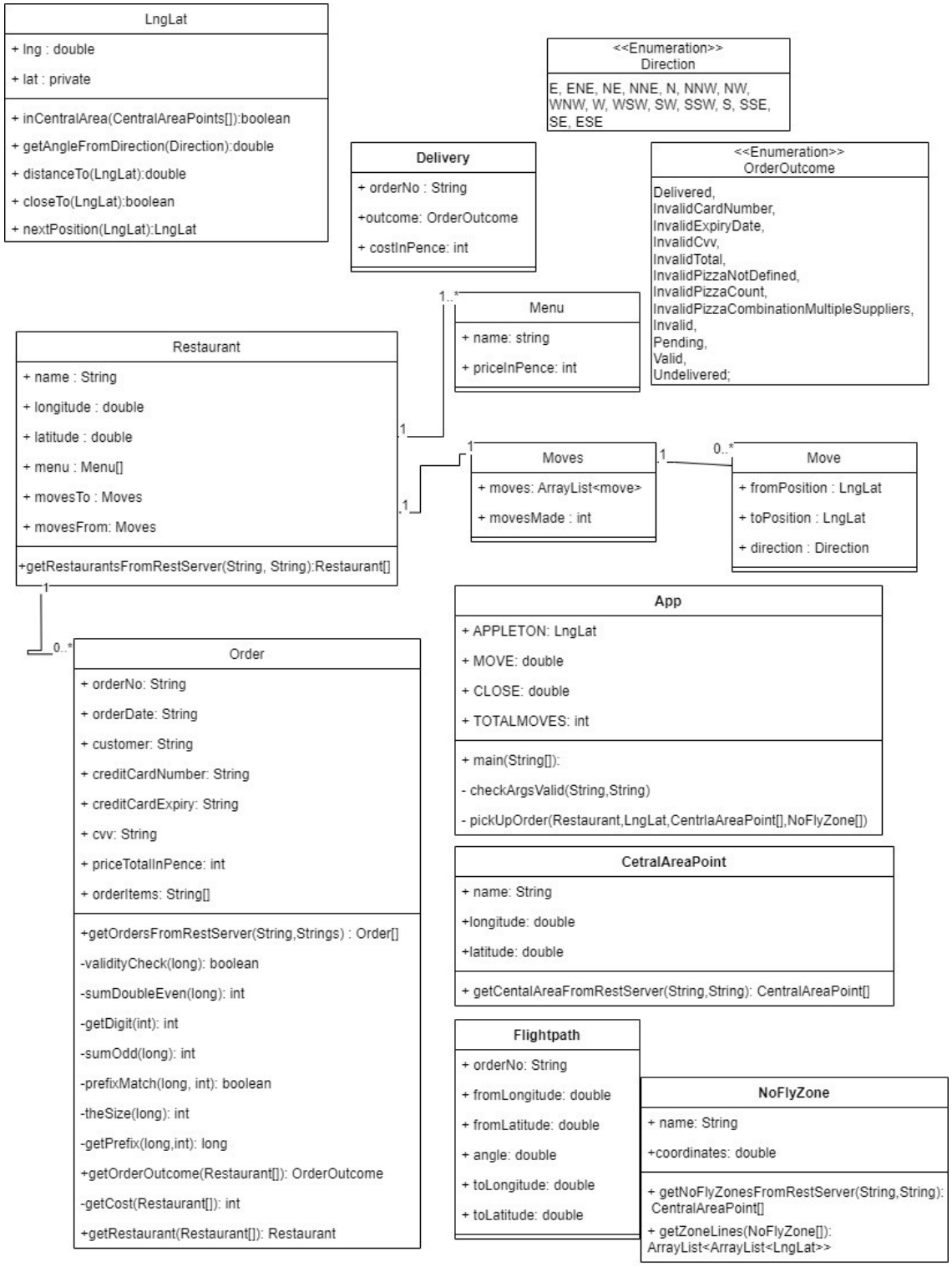
The application needs connection to the internet in order to run as it needs to receive information from the REST API servers on the internet. This is a limitation on the system as the servers could be subject to downtime.

1.5 Use Case Diagram



1.6 UML Class Diagram

In this section I will use a UML class diagram to show the relationship between all the classes and the attributes and methods of each class.



1.7 Class Documentation

In this section I will discuss each class in depth in order to show why I why you identified these classes as being the right ones for your application and describe the functionality of the parameters and method of each class.

App.java

This is the main controller class. The **App** class serves at the entry point of the whole application. This is where the command line arguments are parsed and generates the output files. It has 3 constants that are "public static final" that are used across the classes: APPLETON (the position of Appleton tower), MOVE (the size of a move), CLOSE (the distance in which 2 proximities are considered "close to").

The class contains the following methods:

- **main(String[] args)** This provides the main functionality of the application. It reads in the input arguments and uses the method **checkArgsValid(String date,String baseURL, String random)** to check if the inputs are valid. It retrieves the data from the REST-Server using methods from the classes **Restaurant, Order, CentralAreaPoint, NoFlyZones** based on the baseURL and store them in array of instances their corresponding classes. It selects the order for the chosen date. And calls the methods **pickUpOrder(Restaurant restaurant, LngLat current_pos,CentralAreaPoint[] CA,NoFlyZones[] NFZ)** and **returnAppleton(Restaurant restaurant,LngLat appleton,CentralAreaPoint[] CA, NoFlyZones[] NFZ)** to calculate the flightpath to each restaurant from Appleton and the flightpath from the restaurant to Appleton. It then filters out all the invalid orders from the orders of the given day and sorts them into ascending order based on the number of moves required to get to the restaurant. It then begins calculating the overall flightpath (using the precomputed flightpath to each restaurant) delivering as many orders as possible before running out of moves. It then generates the output .geojson and .json files.
- **checkArgsValid(String date,String baseURL, String random).** This method checks whether the user inputs are valid. By checking the overall length and that the month and date are both integers within the correct range and checks that the date is within the date range that are stored within the current Rest Server. It also checks that the baseURL is correct and reachable. If anything is incorrect then the system exits with a corresponding explanatory message.
- The methods **pickUpOrder(Restaurant restaurant, LngLat current_pos,CentralAreaPoint[] CA,NoFlyZones[] NFZ)** and **returnAppleton(Restaurant restaurant,LngLat appleton,CentralAreaPoint[] CA, NoFlyZones[] NFZ)** work in similar ways. These methods calculates the flight path for the drone from Appleton tower to restaurant passed in and vice versa. It first retrieves the location of the restaurant that it is calculating the path to or from. From the Drones current position it will calculate all possible moves in each possible direction that would be a valid move (i.e. don't go into any no-fly zones or re-entering the central area if has already left using the checkForIntersection(…),inCentralArea(…) methos)and choose the move so that the next move is the one with the closest distance to the desired location(using distanceTo(…)). If there are no valid move, a **LngLat** with junk values (-999.0, -999.0) will be added. It returns each move and direction, and the number of moves made in instance of **Moves** class.
- **checkForIntersection(LngLat currentPos, LngLat moveToPos,NoFlyZones[] NFZ)** This method is used to determine if a move intersects a no-fly zone. It takes the current position of the Drone and the position it wants to go to. It will check if the line between these two

points intersects any line contained within the no-fly zone using the **intersect(…)** method**.** If it does intersect any of the no-fly zones it will return true, false otherwise.

- **intersect(LngLat point1, LngLat point2, LngLat point3, LngLat point4).** This method checks if the line between point1 & point2 intersects with the line between point 3 & 4. It returns true if it does, false otherwise.
- **displayPath(ArrayList<LngLat> path).** This method converts an arraylist of LngLat to a Feature of the type LineString which illustrates the flightpath of the drone. This is used to generate the .geojson file.
- **writeToFile(String fileName, String str).** This method writes given string to given filename.

Direction.java

This is an Enum class which holds the 16 different directions the drone can move in: ***E, ENE, NE, NNE, N, NNW, NW,WNW, W, WSW, SW, SSW, S, SSE, SE, ESE ;*** The values can be accessed throughout the application.

OrderOutcome.java

This is an Enum class which holds all the different outcomes an order can have:

- ***Delivered –*** The order has been delivered
- ***InvalidCardNumber –*** The card number is invalid, checked using Luhn check
- ***InvalidExpiryDate –*** The card exiry date is invalid, either because the month was an invalid (not between 1 and 12) or because the orderDate is the expiry date
- ***InvalidCvv –*** the cvv is invalid because not all integer or is more than 4 digits long
- ***InvalidTotal -*** the provided total doesn't equal the calculated total
- ***InvalidPizzaNotDefined*** – one of the items is not supplied from any of the restaurants
- ***InvalidPizzaCount –*** there is either no pizzas or over 4
- ***InvalidPizzaCombinationMultipleSuppliers-*** the pizzas cant all be supplied from the same restaurant
- ***Pending –*** this is the initial outcome of the order before it goes through any checks
- ***Valid –*** the order is valid but not yet delivered
- ***Undelivered –*** the order is valid but was not delivered as the drone ran out of moves

LngLat.java

This record is used to hold the position of something i.e the drone. The LngLat record contains the following public attributes:

- The **lng** attribute which is a double that represents the longitude of an objects position.
- The **lat** attribute which is a double that represents the latitude of an objects position.

The class has the following methods:

- **inCentralArea(CentralAreaPoint[] currentCA).** This method determines whether a point is in the central area that has been retrieved from the REST server by using a standard point in polygon algorithm.
- **getAngleFromDirection(Direction direction)** This method is used to return the corresponding angle in decimal when given a **Direction.**

- **distanceTo(LngLat position).** This method calculates the distance between the location of the current LngLat and the and the given LngLat using the distance formula distance = $\sqrt{[(x_2 - x_1)^2 + (y_2 - y_1)^2]}$.
- **closeTo(LngLat position).** This method determines if 2 points are closeTo each other, to be used to check whether the drone has reached its destination. It first calculates the distance between the 2 points and then determine whether the distance is less than the CLOSE constant.
- **nextPosition(Direction compDirection**).This method is used to calculate the position the drone would move to when making a move in the given Direction. This is done by calculating the change in longitude and latitude and adding it to the current longitude and latitude. These new values are then be used to create a new LngLat object and the next position of the drone is returned**.**

These methods are used in the **App** class when calculating the flightpath.

CentralAreaPoint.java

The purpose of this class is to hold the information about a point that makes up the central area.It holds its **name**(a public String), **longitude** (a public double) and **latitude** (a public double) and contains methods to get them and set them. It will hold the central area data retrieved from the REST Server.

It has only one method **getCentralAreaFromRESTServer(String baseURL,String relativeURL).** This method reads in the JSON data from a given URL. It checks that the baseURL ends with a / to make sure that the complete URL will be of the correct format. It returns an array containing all the central area points that were received from the REST Server that together make up the central area.

Delivery.java

The purpose of this c class is to represent each delivery that will be written to the Deliveries-YYYY-MM-DD.json file. It has three attributes:

- **orderNo -** a String of the unique order number for the order being described
- **outcome** - the eventual **OrderOutcome** of the current delivery
- **costInPence** - an integer which stores the total price in pence of the current order

Flightpath.java

The purpose of this c class is to represent each delivery that will be written to the Flightpath-YYYY-MM-DD.json file. It has seven attributes:

- **orderNo** -a String that stores the unique order number for the current order being delivered
- **fromLongitude** – a double that stores the longitude of the position that the drone is moving from
- **fromLatitude** - a double that stores the latitude of the position that the drone is moving from
- **angle** – an int that stores the angle at which the drone is moving to its next position

- **toLongitude** - a double that stores the longitude of the position that the drone is moving to
- **toLatitude** - a double that stores the latitude of the position that the drone is moving from
- **ticksSinceStartOfCalculation** - an int that stores the elapsed ticks since the computation

Menu.java

The purpose of this class is to store the information on the individual menu items.Each **Restaurant** uses this class to store its items. It has 2 attributes:

- **name –** a string that stores the name of the menu item
- **priceInPence –** an int that stores the price in pence of the item

Move.java

The purpose of this record is to store an individual move either to or from a restaurant. **Moves** uses this class to store a collection of moves been a restaurant and Appleton. It has 3 attributes:

- **fromPosition** - a LngLat that stores the position the drone will be moving from
- **toPosition** - a LngLat that stores the position the drone will be moving to
- **direction** - a **Direction** which represents which direction the drone is moving

Moves.java

The purpose of this record is to hold all the moves made either to or from a Restaurant and Appleton. Restaurant uses this class to store the path to the position of the restaurant from Appleton vice versa. It has 2 attributes:

- **moves** - an arraylist of **Move** representing all the moves between a restaurant and Appleton.
- **movesMade** – an int that stores the number of moves made to get between the restaurant and appleton

NoFlyZones.java

The purpose of this class is to hold the information about one of the no-fly zones. It holds its **name** (a String)and its coordinates (a 2D array of double) and contains methods to get them and set them. It will hold data retrieved from the REST Server.

It has two methods:

getNoFlyZonesFromRESTServer(String baseURL,String relativeURL). This method reads in the JSON data from a given URL. It checks that the baseURL ends with a / to make sure that the complete URL will be of the correct format. It returns an array containing all the no fly zones that were received from the REST Server.

getZoneLines(NoFlyZones[] NFZ). This method will iterate through all the co-ordinates for each no-fly zone and save them as a LngLat object. Each zones co-ordinates will be saved as a separate ArrayList of LngLat and another arraylist will store all the arraylists of LngLat**.**

Order.java

The purpose of this class is to store information of each individual order that is retrieved from the REST Server. It contains methods to retrieve the orders, determine the orderOutcome, get the restaurant of the order, and the cost of the Order. The methods are:

- **getOrdersFromRestServer(String baseURL,String relativeURL**). This method reads in the JSON data from a given URL. It checks that the baseURL ends with a / to make sure that the complete URL will be of the correct format. It returns an array containing all the orders that were received from the REST Server.
- **getOrderOutcome(Restaurant[] restaurants).** This purpose of the method is to work out if a order is valid, if not set the outcome to the reason itis invalid. Each order starts with its outcome being pending. It begins by checking that the expiry date is valid, by checking that the month is valid and that the expiry date is after the orderDate. If not true the outcome is InvalidExpiryDate.It then checks there is a valid number of pizzas in the order. It also checks the cvv is valid,by checking its length and checks the creditCardNumber is valid by using Luhn check, there is several helper methods used while checking this. Then it checks if the priceInTotalPence is correct by calculating the cost of the order and checking they are equal. Then it checks that all the pizzas can come from the same provider and that each pizza exists.
- **getCost(Restaurant[] restaurants**). This method is used to calculate the total cost of the order in pence including the delivery cost (100). It does this by iterating through the restaurants to find which one the first item comes from. And then iterates through this restaurant to get the price of the other items adding them to the total cost.
- **getRestaurant(Restaurant[] restaurants).** This method is to find which restaurant corresponds to the current order by finding which restaurant has the order items on it's menu.

Restaurant.java

The purpose of this class is to store information of each restaurant that is retrieved from the REST Server. It contains methods to retrieve the restaurant. It also has 2 attributes of type **Moves – movesTo** and **movesFrom** which store precomputed flightpaths used when calculating the full flight path. It has one method:

- **getRestaurantsFromRestServer(String baseURL, String relativeURL).** This method reads in the JSON data from a given URL. It checks that the baseURL ends with a / to make sure that the complete URL will be of the correct format. It returns an array containing all the restaurants that were received from the REST Server.

2.Drone control algorithm

2.1 Algorithm Requirements

The drone control algorithm must meet these main requirements:

- Go to restaurant to collect order and return to the Appleton tower to deliver the order
- Hover at when picking up or dropping off deliveries
- Do not fly through No-Fly Zones
- Once the drone has entered the central area it should not leave again in that delivery
- The drone can make no more than 2000 moves in a given day
- The drone should only collect a delivery if it has enough moves left to also return
- The drone should only deliver the valid orders

The goal of the algorithm is to create a flightpath for orders on a certain date in which it orders as many deliveries as possible while maintaining all the requirements.

2.2 Algorithm Pseudocode

This section will explain the logic of the drone control algorithm using pseudocode.

Main(args)

*…

1. Get the Orders for the given day
2. For each order:
    a. Work out which orders are valid (getOrderOutcome)
3. Add the invalid orders to the delivery array
4. For each restaurant:
    a. Calculate the flightpath to the restaurant from Appleton (pickUpOrder)
    b. Calculate the flightpath to Appleton from the restaurant (returnToAppleton)
5. Sort the orders based the number of moves it would take to collect in ascending order
6. While there is enough moves left to deliver the next valid order
    a. For each move in move to corresponding restaurant for the current order
        i. Add the move to the flightpath array
    b. For each move in move to corresponding restaurant for the current order
        i. Add the move to the flightpath array
    c. Calculate how many moves the drone has left after making completing the order
7. For every undelivered order that is left over
    a. Add this to the deliveries array
8. Write to the flightpath json file
9. Write to the deliveries json file
10. Create the drone geojson file

*The args have been validated and data has already been retrieved. This is has been explained in detail earlier in the report.

pickUpOrder():

1. Get the pickUpLocation(location of restaurant)
2. Set the current location to the location of Appleton tower
3. While the current location doesn't equal the pickUpLocation
    a. Initialise sorted map to store valid moves that will be sorted on the distance to pickup location from the valid point
    b. Initialise sorted map to store valid angles that will be sorted on the distance to pickup location from the valid point
    c. For each direction in the direction enum
        i. Calculate the next position using this direction (nextPosition)
        ii. If the current position isn't in the central area
            1. If the next position isn't in the central area and doesn't intersect with a no fly zone[1] and position hasn't been moved to more than once[2]
                a. Add the move and direction to their sorted maps with their distance to the pickUp location (distanceTo)

iii. else if current position is in central area
    1. If the next position isn't in the central area and doesn't intersect with a no fly zone and hasn't already been made[1]
        a. Add the move and direction to their sorted maps with their distance to the pickUp location (distanceTo)

d. If there is no valid moves
    i. add the junk value to current position

e. else
    i. store the best move as bestMove
    ii. store the best direction as bestDirection
    iii. store the currentPos, bestMove, bestDirection as a Move object
    iv. add the move to arraylist of moves

4. if current position is near to pickUpLocation
    a. add hover move

The method returnToAppleton works in primarily the same way as pickUpOrder. The only exceptions are c.ii would instead be

c

ii. If the current position is in the central area

1. If the next position is in the central area and doesn't intersect with a no fly zone and hasn't already been made[1]
    a. Add the move and direction to their sorted maps with their distance to the pickUp location (distanceTo)

ii. else if current position isn't in central area

As if it is going from the restaurant to Appleton tower then once in enters the central area then it should leave again.

Clarificatons

1. The algorithm instead of just checking that the next position would be in a no-fly zone also checks if the line of the move would intersect with any of the no-fly zone boundarys.

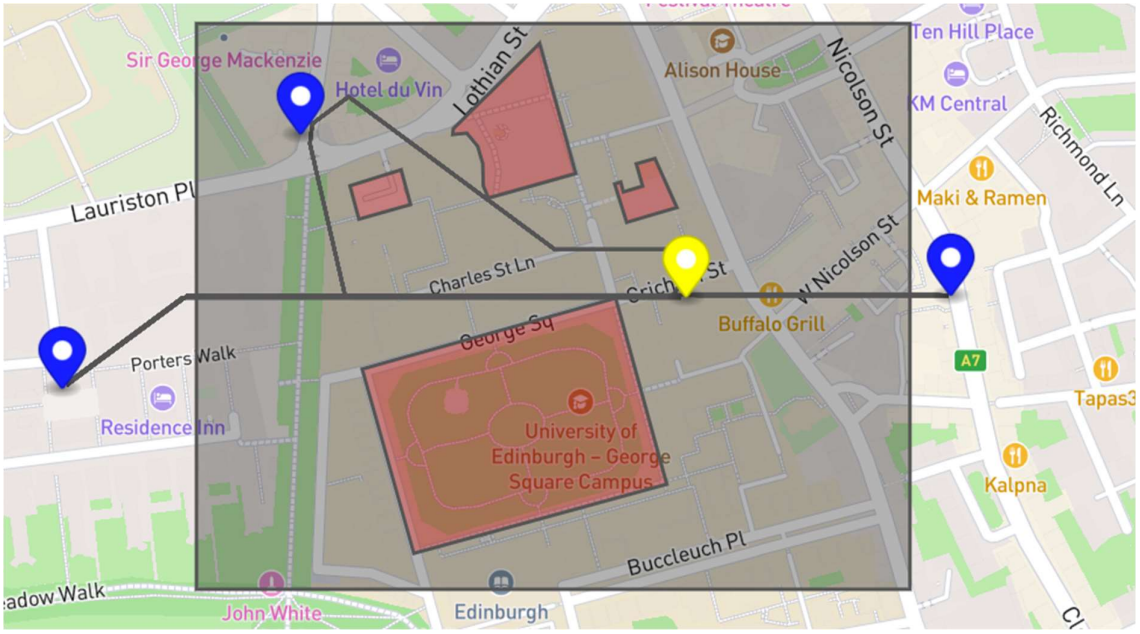
2. The moves previously made are stored in order to check so that we can check if the move has already been made. This is to avoid the drone getting stuck between 2 positions if it is at a boundary or a no fly zone and instead will attempt a different route. This is in case a situation occurs where the drone is surrounded by no-fly zones when trying to get to its desired destination. In this situation it would try moving away from the no-fly zone in the direction it previously came from but then move back to where it was as it would think it was getting nearer to the desired location. In order to prevent this we store the moves that have been made, in order to check if the drone has already been at said position. I allowed the drone to visit the same position once in case it is stuck in a position where the only ideal move is to go back to its previous position.

Summary

The basis of the pickUpOrder() and returnToAppleton() methods are that the pick which move to next make based on which direction will bring the drone closest to its pick up location. These are used to calculate the route to and from each restaurant. The main method prioritises the orders that will require the least amount of moves to be made in order to maximise the number of orders delivered.

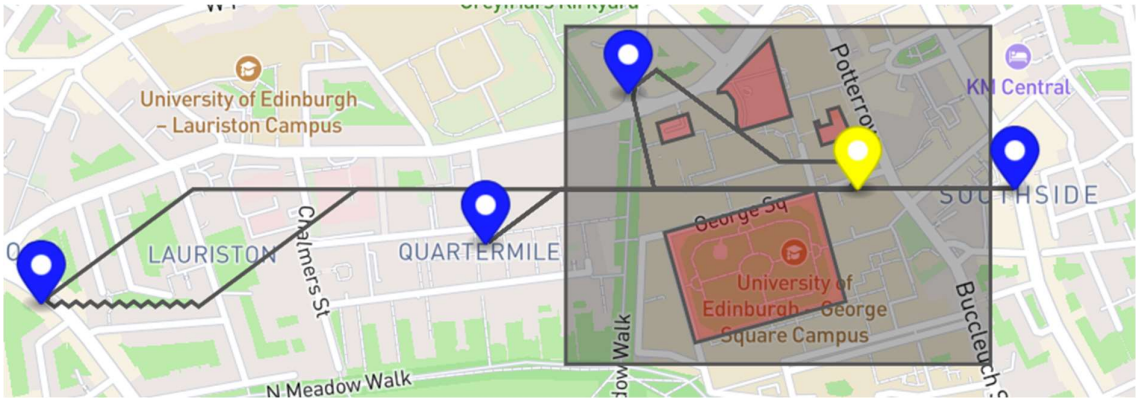
GeoJson Examples

Drone-2023-01-01



On this day the drone delivered 26 orders.

Drone-2023-01-14



On this day the drone delivered 26 orders.