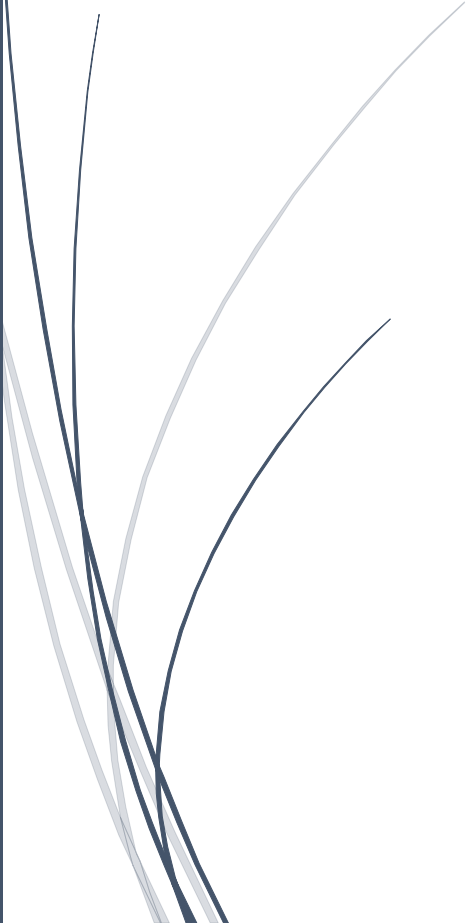


A dark blue vertical bar runs down the left side of the page. A blue arrow points to the right from this bar, containing the date.

12/9/2022

ILP-Report

PizzaDronz Project

Several thin, curved lines in dark blue and light grey originate from the bottom left corner and sweep upwards and to the right.

Ashley dong
S2015258

Introduction

This coursework involved taking pizza orders from the rest server for validation, and finding the drone flight paths for those valid orders. We try to maximise the orders that could be delivered while avoiding the no fly zones given to us.

This report will take you through the choice of classes, the choice of the drone path finding algorithm, the results of how many average deliveries were delivered based on the flightpath taken, and the visual map of the final flightpaths for this implementation of the coursework.

Part. 1 - Software Architecture Description

Coursework one allowed me to treat coursework two as a way to make and combine building blocks to construct something larger. A lot of code was reused from coursework one to serve a foundation to make the final project implementation result possible.

There are 18 total classes in the implementation of this project. They are all interconnected and used through the main *App* Class. Each class are brought to use by the main *App* class, and are necessary for the whole project to work. I will be explaining why the classes are chosen in each of their own sections.

These classes all have a particular purpose, but there are 4 main categories of what the classes purpose lies in. The first is handling the deserializing of the Json data after retrieval from the rest server, second is the validation of the orders from the rest server, third is calculating the drone flight path, and the last is to convert our results to Json and GeoJson files.

UML Class Diagram

Please see figure 1 for the visualisation of the classes as a UML diagram.

From the diagram, we can see that the main *App* class is the root of all connections in communicating between the class and their instances.

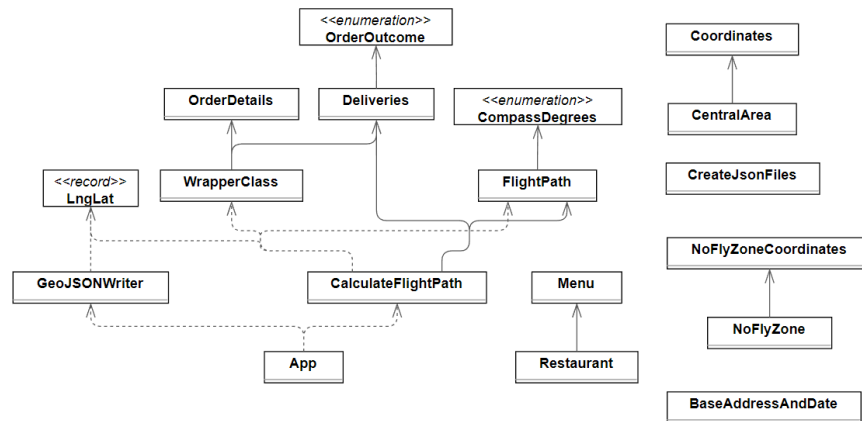


Figure 1

App

The main class for our project. The main method here constructs the necessary steps for creating our 3 files for a given day. The App class reads the user inputs for the order date and the rest server base URL address to be used throughout the classes. The steps taken to create the files are very clear.

The first step is to create an array (*OrderDetails[] orders*), which contains all the orders details for that day. Another array (*Deliveries[] inCompleteDeliveries*) was created using the orders object to validate orders using their details provided. We do not yet know what orders can be delivered yet, which is why the array is called *inCompleteDeliveries*.

The flightpath and deliveries are then created using the previous initialized objects as fields of the object of the *CalculateFlightPath* class and their files are written converted to Json and GeoJson using the necessary classes.

This class is very important to bring together all the classes and use them in a simple way to create the final files required.

CalculateFlightPath

This class does all of the calculations for the path of the drone, and also updates the incomplete deliveries outcomes for orders that have been delivered. It contains two field members, flightpaths and deliveries, which is initialised in the constructor method of this class. In fact, the constructor method contains all the calculation needed for the flightpath of the drone and the delivery status.

I decided to do all my calculations in the constructor method with helper function due to needing to update and return two variables at the same time. It is easier to store the finished calculations as field variables which you can access from the object instance.

LngLat

This is a record class that contains the coordinates of any object created by this class. This is really useful to be able to use the methods that is contained within the class on any *LngLat* object. It is especially useful in creating a flightpath as a point can be easily created for checking certain requirements.

It is useful for checking for if a point is in certain area, it's own position and another point is intersecting a zone or not, all needed for the *CalculateFlightPath* class.

OrderOutcome

This is an enumeration class that specifies all the possible allowed order outcomes that an order is allowed to have, which is why an enumeration class was chosen. When orders are getting updated they can only have one of the 10 outcomes.

CompassDegrees

This is an enumeration class that contains immutable constants. It contains the major 16 compass degrees (see figure 2) that the drone is allowed to move in according to the specifications. This ensures that the value of the angle used is always one of the 16 states that is specified. It still contains a getter that allows you to obtain the value of the *CompassDegree* angle. This allows you to perform calculations on the angle but still ensures that other methods and can only use the enumeration values that this class provides to enforce drone movement.

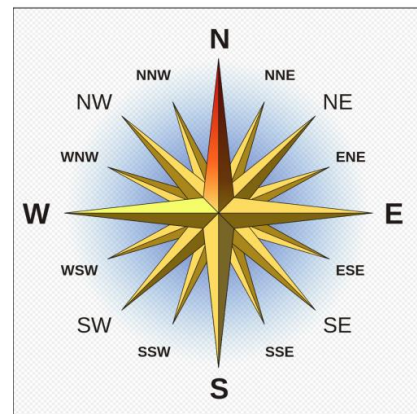


Figure 2

Deliveries

This class contains the field members required for the Deliveries class specified in the specifications. The constructor class sets the field parameters from the arguments given to it directly. This class also creates and validates all the orders when the *getDeliveries()* function is called. The method contains else if statements checking over all details of the orders. It then return an almost validated but not tried to be delivered yet order outcome array *Deliveries[]*.

I found that having the method for validating the orders in the same class as creating the *Deliveries[]* array was much more useful and simpler than creating another class.

Coordinates

This class is used to store Coordinate items when retrieved from the REST server. It contains the longitude and latitude and the name of the point.

Menu

This class is used to store the Menu items when retrieved from the REST server. It contains the name and the cost of each pizza.

Restaurant

This Class deserializes the Json object into Restaurant objects that contains the name, array of Menu items and the longitude and latitude of the restaurant. It has a getter that returns the list of participating restaurants fetched from the REST server. It also provides a function that finds the selected restaurant based on the list of the order items provided, which is used in another class.

FlightPath

This class is used for storage only, and is created according to the coursework specifications to write a Json file using the stored values. Therefore the constructor stores the arguments given straight to its field constructors.

WrapperClass

This class was created just for the use of one function in the *calculateFlighPaths* class. As I wanted to sort and return two arrays based on the information from one array. This wrapper class allowed me to return two different typed variables that saved runtime and gives a easier implementation for sorting two arrays at once in a function. It contains a constructor method to set the field member of the object instance which is public and easy to access.

BaseAddressAndDate

This class contains two getters and a setter for easy access to the user input dates in all classes that require it, mostly used to deserialize the Rest server information. This class also contains a method to check if the user input values for the date and URL are correct or not.

NoFlyZone

This class is another class that is used for deserializing json data from the REST server to get the corresponding *NoFlyZoneCoordinates[]* array that is used elsewhere in other classes.

NoFlyZoneCoordinates

This class stores the *double[][] coordinates* and the name of the coordinates of the no fly zones items when retrieved from the REST server.

CentralArea

This class is very similar to the *NoFlyZone* Class where it used to deserializing json data from the REST server to get the corresponding *Coordinates[]* array that is used elsewhere in other classes.

OrderDetails

This is another retrieval and deserializing class that gets data from the REST server and converts the Json data to an array of *OrderDetails* in the *getOrderDetails()* method.

CreateJsonFiles

This class creates the required files *flightpath-YYYY-MM-DD.json* and *deliveries-YYY-MM-DD.json* files in the right folder. The arguments are given by the main App class that created the flightpaths and the completed deliveries.

GeoJSONWriter

This class creates a geojson file that contains the drone flight path. It uses the *ArrayList<FlightPath> flightPaths* argument give to create the required *LngLat* List of points used to make the geojson file. It also creates the resultfiles folder in the right root directory if the folder does not exist.

Part. 2 - Drone Control Algorithm

I will now explain the drone algorithm in this section of the report and provide results on the average sample deliveries made in a day. The methods used to compute if the drone is making a valid move or not will also be explained by each constraint placed on us. Figures will also be provided for two random days' flight path overlaid on the university campus's no fly zones and central area.

The Greedy Algorithm

The approach taken was a greedy algorithm where the path taken is the one where it is the closest to the destination out of all the possible valid moves left, which is in the *pickBestMove()* method in *CalculateFlightPath* class. It makes sure that it will not go back in the opposite direction it took in the previous step. However, path finding was only done for the flight path calculation movement to the restaurant. The path back to Appleton Tower is found by using the path taken to the restaurant and flipping the angle of travel in the opposite direction to go back to Appleton Tower, this saves computation time and ensures that all the paths to each restaurant is the same.

The algorithm keeps trying to deliver if it still has moves or deliveries to be delivered left over. When moving to the restaurant for a particular delivery, we will keep picking the best valid move in a greedy approach until we are close enough to the restaurant coordinate.

Once we arrive, we will use the counter *movesToRestaurant* to use the same amount of moves to move back to Appleton Tower.

The drone does do hover moves before moving to a new location, although the first hover is not needed at Appleton Tower, it was added to start the algorithm. The initial hover mover is then deleted at the very end of the algorithm

Maximising sample deliveries per day

Using the 12 files I have generated, the average deliveries per day is 29 per day. This is because I had sorted the deliveries so the orders that go to the closest restaurant is prioritised over the further away ones. This does seem unfair to the customers who wanted their pizzas from the further away restaurants though, as it means that they would never usually get their meal. However, delivering to the closest restaurants first did allow me to deliver more orders. When sorting by distance of restaurant was not implemented, the average sample orders would only result to 17 orders per day.

The use of the Greedy Algorithm could really be improved, as another algorithm such as the A* algorithm would be a better option to finding a more efficient path. The greedy algorithm may choose to go the most inefficient way around an area if the slope is slanted slightly in an unfavourable angle, whereas A* would most likely not be affected in this way.

Sorting By Shortest Distance

The sorting algorithm used for sorting by shortest distance restaurant orders is bubble sort. This allowed me to sort my deliveries and orders at the same time, sorting whenever the selected pizza restaurant (using the *findRestaurant()* method) distance is larger than the one before. A wrapper class was needed to act as a way to return two different variable types, namely, the sorted deliveries and orders by orders with closest restaurant. It was then easily accessed in the main calculating flightpath constructor method.

Constraints

In the *PickBestMove()* function, there are several constraints that are set so that a certain move is not picked. Firstly, the drone after moving must not end up inside a no fly zone. Secondly, the drone flight must not cut over a corner of the no fly zone. Thirdly, the chosen direction must not be in the opposite direction of what it is traveling in before, therefore traveling back on itself and lastly, once back inside the central area, it cannot leave again until the pizza has been delivered to Appleton tower.

I will explain the approach to finding if the drone has made a valid move or not.

Not in a No Fly Zone

The method *inNoFlyZone()* in the *LngLat* class can be called on the *LngLat* point in question. This method returns true if the point is in one of the no fly zones. For each of the no fly zone polygons described by its coordinates, we use the *isInPolygon()* method to check if it is in that polygon. The way the *isInpolygon()* method works is that it uses ray tracing method to

draw a line from the point along the x-axis to infinity and counts how ever many times it passes through an edge. If that number is odd, then the *LngLat* point will be within the polygon, otherwise it is not within the polygon. We have to use this method because the *noFlyZones* are not regular rectangles and can have convex characteristics. This method is heavily influenced by a stack overflow forum¹.

Not cut a corner in its flightpath

The method *cutsNoFlyZone()* in the *LngLat* class can be used to check the current *LngLat* point and the next *LngLat* point flight line of path will intersect with a *noflyZone*. This method creates a line with the *Line2D* class for the test drone flight movement and compares with all the no fly zones lines. The lines are created by using the adjacent coordinates for each of the no fly zones.

Must not travel back on itself for one move

The *oppositeDirection()* method in the *CalculateFlighthPath* class gets the double value of the angle provided and adds 180 to it. Then the modulo of 360 of the answer is taken and the corresponding Compass degree is returned. There is an if statement to check if the current direction the drone is considering is that opposite angle and stops the drone from moving in that direction if the angle is the opposite angle. This also helps prevent the drone going back and forth when encountering a wall.

Back in Central Area

This is done with the path finding to the restaurant, I make sure that once the drone has left the central area using a Boolean variable to keep track of when it leave the central area, that it will not go back in again. Checking using the *inCentralArea()* function which also uses the *isInPolygon()* function. This is because since the drone takes the same path back, it will only go inside central area and will not leave it again.

Part. 3 – figures from GeoJSON Flight Path

We will lastly include the GeoJSON figure of our drone flight path from two separate days:

2023-02-04:

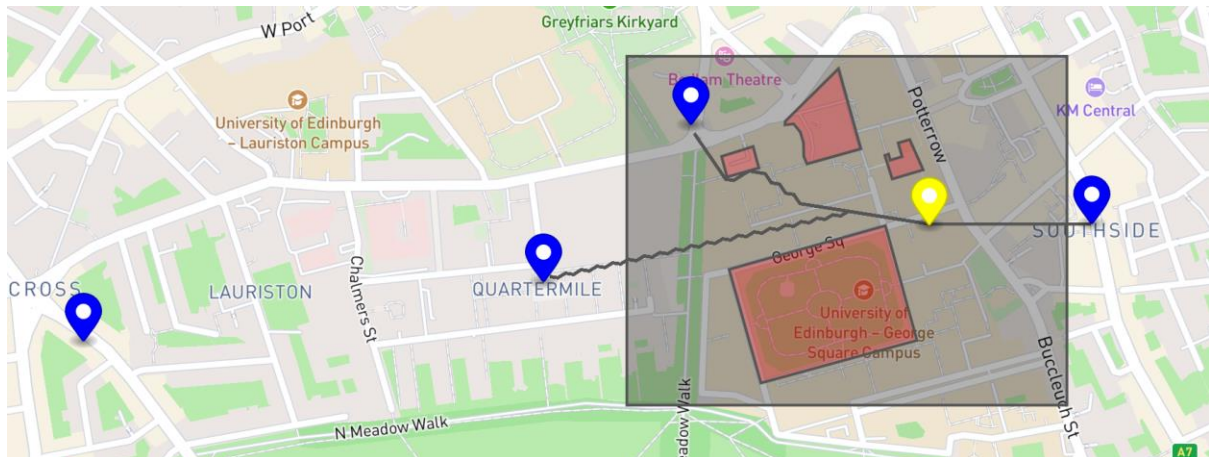


Figure 3

2023-03-11:

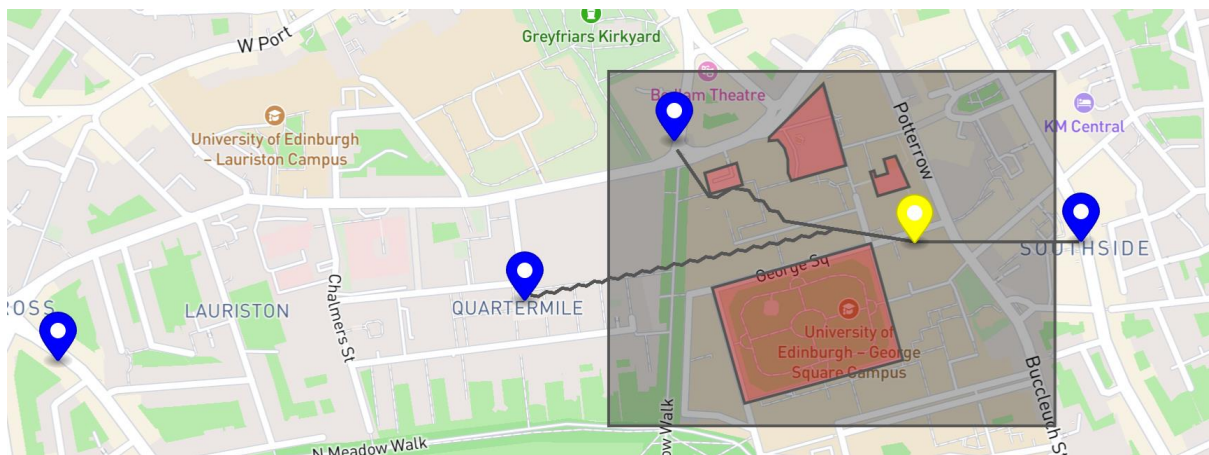


Figure 4

We can see from the above images that the drone does not deliver the furthest away restaurant at all, as it runs out of available moves delivering orders that are for the closer restaurants.

References:

1. <https://stackoverflow.com/questions/8721406/how-to-determine-if-a-point-is-inside-a-2d-convex-polygon> Accessed Online at: 13/08/2022