UNN: ***(s1918278)***

Course Name: Informatics Large Practical

Course Organiser: Stephen Gilmore and Paul Jackson

Submission Date: 03/12/2021

Informatics Large Practical

Coursework 2

Report

Section1: Software Architecture

Introduction:

This section will explain the needs of each class when designing this software to meet the purpose of making deliveries in terms of each class's designed-purpose and their uses in other classes.

1.0 LongLat

LongLat is needed to define a coordinate on a geographical map by providing the longitude and latitude of the location in double-precision. This is a very basic class and is used frequently in other classes when a location is needed to be specified. Methods in this class are capable of checking if the location declared is within the confined area as well as whether it's within a move to a given location. Also, the class contain methods to calculate the distance, angle and number of moves between the location declared and a given location. Based on the angle between two locations, method in this class is also able to calculate the drone's next location on the map.

1.2 DataBase

DataBase class is declared when the program needs interaction to the database. This class is able to establish a connection to the derby database server by constructing a JDBC string in the class. Based on this connection, methods in this class are able to read tables stored in the database – **"orders"** and **"orderdetails"**, as well as create and write tables – **"deliveries"** and **"flightpath"** in the database. The results obtained from reading the tables in the database is used in *Drone* class when there's needs to obtain a list of valid orders to deliver on the date specified.

1.3 WebAccess

WebAccess class is declared when the application needs content stored in the web server. This class would initiate a **HTTP client** when it's declared. With the use of this client, a method would send **HTTP request** based on the file location to the webserver and return the resultant **HTTP response** as a processable JSON string. The JSON string obtained would be either parsed using GSON parsers or be parsed as a collection of Features in different classes depending on the folder of the file located in.

1.4 W3wordDetails

W3wordDetails class is used as a GSON parser for the content in each w3words location file stored in the *"words"* folder. It contains a method to return the coordinate of the w3words location as an instance of *LongLat* class. This class is called later in *W3word* class to parse the JSON string obtained from files located in the "*words*" folder.

1.5 Shop

Shop class is used as a GSON parser for the content in ***"****menus.json****"*** file stored in the *"menus"* folder. An instance of this class represents a single shop in this application. Methods of this class are able to pack the shop's items' prices and location information in HashMaps with items name as keys. Those HashMaps are later used in *Menus* class when making an aggregated HashMap which stores information of all the items available to order from the application.

1.6 W3words

W3words class is used to translate a location in w3words format to a geographical coordinate as an instance of *LongLat* class frequently in the methods defined in *Drone* class.

1.7 Menus

Menus class offer access to information related to every item appear in orders. Methods of this class are able to create aggregated HashMaps over every items for the purpose of finding an items' price and shop location using the item's name. Those maps are used in methods under *Order* class to calculate each order's value and summarize each order's shop locations needed to travel by the drone for items pick-up.

1.8 Buildings

Buildings class is needed to organize the information stored in the files located in the *"buildings"* folder (*"no-fly-zones.geojson"* & *"landmarks.geojson"*) by paring the JSON string obtained using an instance of *WebAccess* class in to a collection of *Feature* objects. Methods in this class are able to generate the borders of the no-fly-zones and the coordinates of the landmarks based on the collection of Features for the needs of checking on no-fly-zones later in the *Drone* class.

1.9 Order

Order class is needed to process and store information related to an order. Local fields: **orderNo**, **deliverTo** and the integer returned by the method *getOrderCost* formed an entry in the output "deliveries" table in the database. Instances of this class are also declared in *DataBase* class to store the order information obtained from the databases on a given date as a collection of Orders.

1.10 FlightPath

FlightPath class is needed to store information required in the output "**flightpath**" database table. The local fields: **orderNo**, **fromLongitude**, **fromLatitude**, **angle**, **toLongitude** and **toLatitude** form an entry in the table which represent a single flight path. An instance of this class is declared in *Drone* class to store the flight path made by the drone when it takes a single move in a given direction during the process of making deliveries.

1.11 Drone

Drone class is needed as it contains the main algorithm to generate and store the flightpath needed for the drone to follow on a specific date. It also contains helper functions to support this main delivery making algorithm.

To ensure the drone won't cross no-fly-zone while it's delivering orders, methods in this class will perform checks on the drone's next flightpath on the collection of no-fly-zone borders. If the drone's next flight path does cross, the method would re-plan the drone's next visit to one of the two landmarks. Information regarding the no-fly-zone borders and landmarks are obtained using instances of *Building* class. And they are declared as local fields in this class since checks on flightpaths need to be done frequently.

In order to maximising the percentage monetary value of making this day's delivery. Methods in this class will return the order that carries the highest "value-move" ratio from a collection of undelivered orders as the drone's next order to deliver. The call on an order to get an order's cost in calculating the "value-move" ratio involves a declaration of an instance in *Menus* class.

1.12 Result

Result class is needed to output the resultant flightpaths into a GeoJSON file as well as create and write the two database tables "deliveries" and "flightpath" into the database.

1.13 App

App class is needed to take-in user's input for program execution. The inputs are used to declare an instance of *Drone* class in *App*. After calling the algorithm of making the day's delivery on this drone, the drone would be passed to declare an instance of *Result* class to output the results.

Section2: Drone Control Algorithm

Introduction:

This section would first declare the problem tackled by the algorithm: *makeDelivery()* in the *Drone* class and state the assumptions made for this algorithm. Followed by a detailed explanation of the algorithm's strategy in planning the drone's flightpath in terms of how the drone avoid crossing the no-fly-zones and always remains in confined area; how the algorithm selects the next order for the drone to deliver in order to maximise the day's monetary percentage value within the limited moves; how the algorithm guarantee the drone's return to the Appleton Tower at the end of the days' delivery and how the algorithm store the flight paths and orders made by the drone which needs to get outputted at the end of the program. Two example flightpath illustrations will be displayed after the detailed explanation. And the section ended with a brief discussion on the algorithm's runtime and efficiency.

Problem:

By providing a date, the drone is asked to deliver the orders made before the day's noon (11:59). From 12:00, the drone will be departed from Appleton Tower to make the deliveries follows the flightpath generated in the second between 11:59 and 12:00.

The *makeDelivery* algorithm needs to take a date as unput, plan the route and output the resultant drone's flightpath within one second. The flight path made should avoid crossing no-fly-zones while trying to make as much orders as possible within a move limit of 1500 and trying to maximise the day's percentage-monetary-value. And each day's flightpath should be ended at a location that's close to Appleton Tower.

Assumptions:

1. This algorithm assumes all shops and deliver-to locations are located within the confined area.

2. This algorithm assumes that if under an order, there are multiple items need to be collected from the same shop. The drone only needs to travel there once for pick-up.

3. This algorithm assumes there's always going to be at least one valid landmark the drone can take detour to avoid crossing no-fly-zone.

Strategy:

1. How to plan the flightpath:

Every flight path made by the drone must not cross the no-fly-zone and the drone must always remain in the confined area.

1.0 How to avoid crossing no-fly-zone:

The drone can't fly into the no-fly-zone during the entire process of making the deliveries of the day. Hence, before the drone start taking any move to its next destination, the algorithm would check if the direct path from the drone's current location to the destination would cross any of the borders of the no-fly-area in a ***heuristic*** approach.

- *If cross:* the algorithm would instruct the drone to travel to one of the two landmarks before visiting the destination in order to avoid crossing no-fly-zone. By first filter valid landmarks -- neither the path from the drone's current location to the landmark nor the path from the landmark to the destination would cross the no-fly-zone. Then pick out the one that's located closest to the drone's current location.
- *If don't cross:* the drone can travel to the destination directly.

1.1 How to ensure the drone is within the confined area:

The drone has to remain in the confined area during the entire process of making the deliveries. Hence, in the step of updating drone's current location to its next location. The next location has to be checked if it's within the confined area.

- *If yes:* update the current location to the drone's next location.
- *If not:* The console would report the detection of invalid location and the system would be terminated.

2. How to select the next order to deliver:

Due to drone's moves limit, there's no guarantee that the drone is able to deliver every single order each day. In order to maximise the percentage monetary return of the day, the algorithm would let the drone to deliver the orders in a ***greedy*** approach. By picking out the order that has the highest "value-moves" ratio from a collection of valid orders that's undelivered yet. The "value" term refers to the earning made from delivering an order, and the "moves" means the number of moves needed for the drone to complete the delivery journey of this order.

3. How to guarantee drone's return to the Appleton Tower:

There are two scenarios when the drone would return to its starting point – the Appleton Tower:

I. *The drone finish all of the day's delivery orders:* The drone will return to the Appleton Tower after finish delivering the last order.

II. *There are still orders left undelivered, but drone doesn't have enough moves for completing all of them:* The algorithm would loop through

the collection of undelivered orders to search for the order that's able to be delivered:

- *If no such order found:* Drone would return to Appleton Tower from its current location.

- *If such an order is found:* The drone would complete this order and return to the previous move checking process to see if there's any order can be made.

4. How to check if the drone has enough move left for completing the next order's delivery:

In order to make sure the drone always have enough moves to return to the Appleton Tower in each of the case scenario mentioned above. The number of moves left in the drone after the delivery of the next order is compared to the moves needed to travel to Appleton Tower from the order's deliver-to location before an order is made.

- *If the move left is enough*: Drone would start deliver this order.

- *If the move left is not enough*: Drone would give up this order by removing it from the list of undelivered-orders and back to the move check on the rest of the orders.

5. How to store the flight path and orders made by the drone:

In order to store information needed for the two output database tables and one GeoJSON file, three local fields are declared in Drone class: **orderDataBase**, **flightPathDataBase**, **geoJsonList**. And they will be updated continuously during the process of making deliveries at different stages:

- **orderDataBase**—Orders made by the drone: updated when the drone is close to the order's deliver-to location.

- **flightpathDataBase**—Flight paths made by the drone: updated when the drone makes a move.

- **geoJsonList**—Coordinates visited by the drone: updated when the drone makes a move.

By following the strategy, the algorithm *makeDelivery* is able to plan and store the flight path for the drone to follow in making the day's delivery. As a result of testing the algorithm on 12 dates, twelve GeoJSON files are generated and the contents in two of them are displayed on the next page.

Example Flight Path:

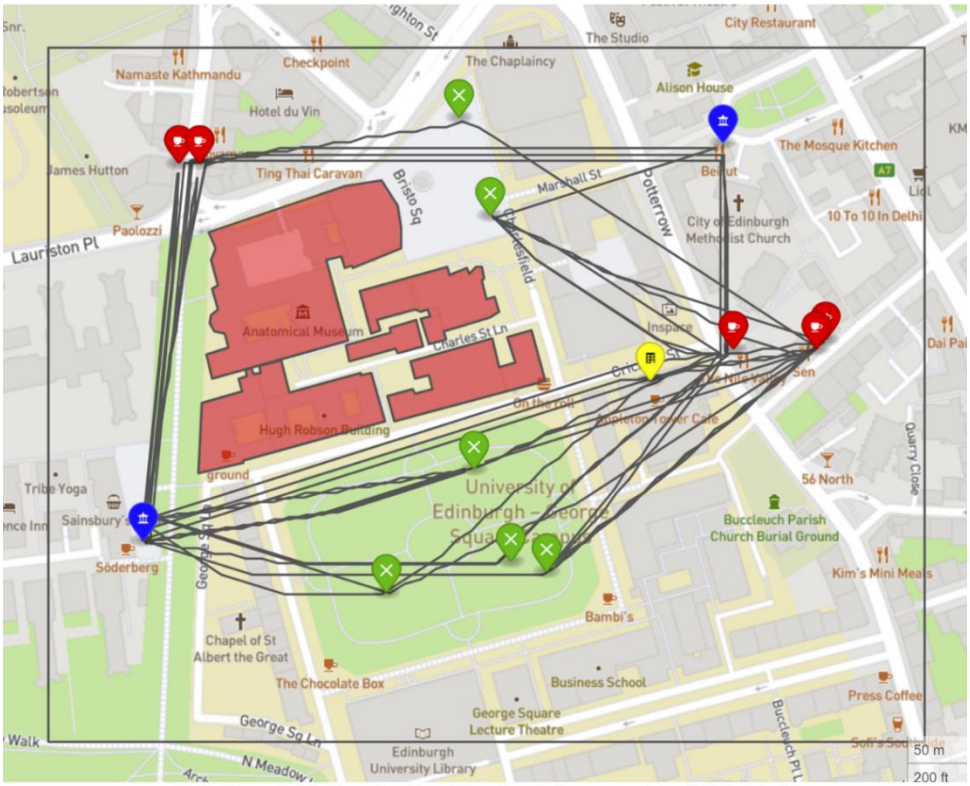

Fig1: Drone's flight path of making the delivery on date: 2022-12-12.

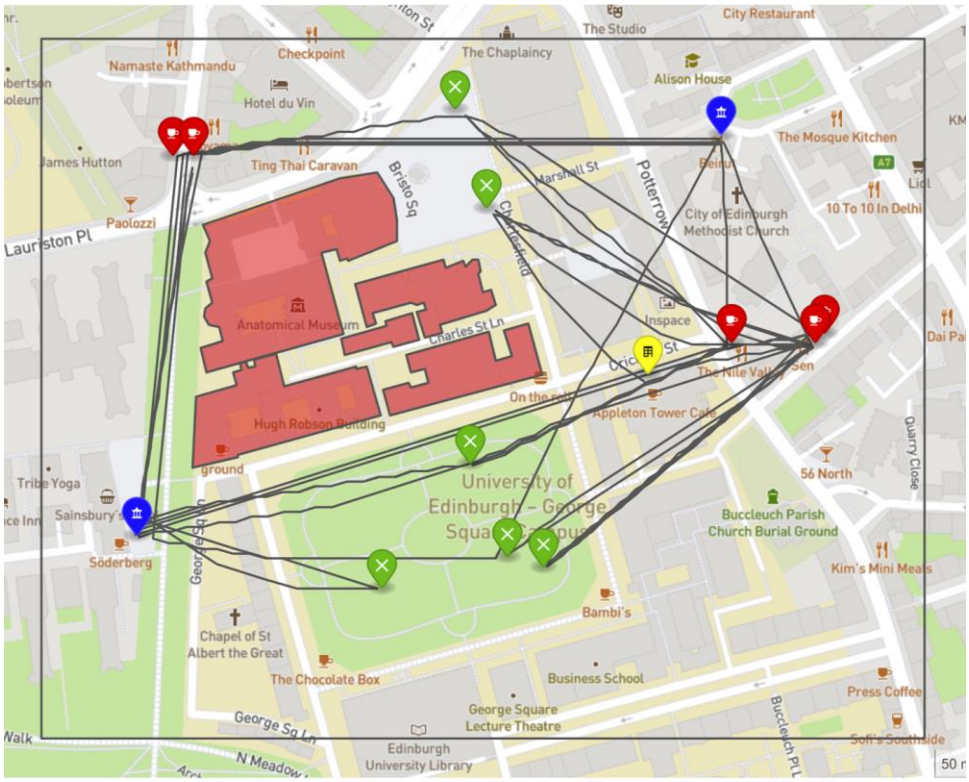

Fig2: Drone's flight path of making the delivery on date: 2022-09-09.

Run time and efficiency:

The table display the performance of the algorithm when testing on twelve dates in 2022:

Date	Move Left	Percentage-Monetary Value	Run Time /Seconds	Date	Move Left	Percentage-Monetary Value	Run Time /Seconds
01-01-2022	1168	100.0	0.592	01-01-2022	701	100.0	1.454
02-02-2022	1181	100.0	0.546	01-01-2022	556	100.0	1.863
03-03-2022	1162	100.0	0.611	01-01-2022	705	100.0	1.780
04-04-2022	1167	100.0	0.706	01-01-2022	612	100.0	2.164
05-05-2022	927	100.0	0.943	01-01-2022	313	100.0	2.339
06-06-2022	929	100.0	0.931	01-01-2022	460	100.0	2.285

Table1: Flight path generation performance on 12 dates in 2022.

The average run time for generating the twelve testing files is around 1.355 seconds. The algorithm is able to finish every order on those dates hence achieving 100% (maximized) percentage-monetary value.

However, as the testing went on to some dates in year 2023, there exist cases when the percentage-monetary value can't be maximised. For instance, when testing on 5 dates in 2023:

Date	Move Left	Percentage-Monetary Value	Run Time /Seconds
08-08-2023	108	88.78	5.859
09-09-2023	56	87.37	5.205
10-10-2023	83	83.84	5.267
11-11-2023	56	87.37	5.402
12-12-2023	108	88.78	6.120

Table2: Flight path generation performance on 5 dates in 2023.

The average run time is around 5.571 seconds, and the average percentage monetary value is about 87.23%.

Hence the algorithm is atill able to finish flightpath planning and generating within 60 seconds despite it can't maximise the percentage monetary value when there's large amount of orders to deliver.