Informatics Large Practical

Implementation Report

S1956488

1.Software Architecture Description

In this section, I will provide a description of the software architecture of the drone's program. I will provide an explanation for why I had chosen certain Java Classes as the right one for my implementation .

1.1Reasoning behind the classes

The **App** Class is the entry point for the application whereby here is where the input arguments are parsed and checked to ensure that they are valid and not null. The **App** Class is also where the methods from other classes are called such as getCoordinatesAndOrders() from the **Orders** Class and getNoFlyZones()from the **GeoJsonParser** Class. The number of moves that the drone could have before the battery dies is also initialised here when initializing the **Drone** object. The getRoute() method is called in this class and the results are stored in a list of **LongLat** objects which will then be passed to movesToFCCollection() method to get the json string of the feature collection. This is then sent to the outputGeoJsonFile() method to create the geoJson file and the two databases (deliveries and flightpath) are created and written to.

The **Drone** Class is to create a drone object that would be contain the number of moves it is able to make before the battery dying. I decided to make a drone class and add this value there instead of hard coding it into the program as I wanted the developer to be able to change the number of moves easily if the drone gets upgraded or downgraded. All that the developer would have to do is the change the number of moves that is passed as an argument when initialising the drone object in the **App** Class

The **Deliveries** and **Flightpath** class represent the classes I used to parse the data before adding it to their respective databases.

The **WordsDetails** class was used to parse the json string received when reading the WhatThreeWords files and to get the coordinates of the location

The **Menu** Class was used to parse the json string received when reading from menus.json file in the webserver.

The **ApacheDatabase** class was created to make it easier to parse the data that needs to be added to the deliveries and flightpath tables and also to create those tables.

The **GeoJsonParser** class was created to mainly read and write geojson files. It would read the geojson files in the webserver and also parse the movements of the drone to a Json string of its feature collection and then write it to a newly created geojson file.

The **HexGraph** class was used to create the hexagonal graph that the drone would use as a guide to travel on and also to find the shortest path for the drone to travel on the graph. Various TSP algorithms were also implemented here to find the optimal flightpath.

The **LongLat** Class was created to represent a location on the map

The **Menus** class parses the menus file from the webserver and also contains the methods that allows the program to fetch the price of items and calculate the cost of an order

The **NodeEdges** class is just used to represent an edge on the hexagonal graph that was created.

The **Orders** Class was created to get the information needed to complete an order such as the order number, the delivery locations and the pick-up locations of the shops.

1.2 Class Documentation

In this section I will give an in-depth explanation to each class and explain the main methods and variables of the classes

App.Java

The main controller of the program. It accepts the inputs and checks if they are null. It has the following variables that are public and static as they are accessed by other class:

- day
- month
- year
- webServerPort
- databasePort

The class does not contain any methods, but it does call upon methods from other classes which are as follows:

- Orders.getCoordinatesAndOrders() – which runs the methods in the Orders class
- GeoJsonParser.getNoFlyZones() and getLandmarks() – which saves those features in variables
- It initialises a new drone object
- HexGraph.getRoute(drone) - this a list of all the moves by the drone when delivering the orders
- GeoJsonParser.movesToFCCollection(moves) – this returns a json string of the feature collection of moves by the drone
- GeoJsonParser.outputGeoJsonFile(fc) – this method creates and geoJson file in the current working directory and writes the json string to it.
- ApacheDatabase.createDeliveriesDatabase() and createFlightPathDatabase(moves) creates the 2 tables that records the orders delivered and the flightpath of the drone.

ApacheDatabase.Java

The class for parsing data and writing to the deliveries and flightpath table. It has only one variable which is private:

- jdbcString – the string used to access the database and is fed as an input to the getConnection function.

The class has the following main methods:

- getDeliveryList() – the method creates a list of Deliveries objects which will then be used when adding it to the database
- createFlightPathsList (List<LongLat> moves) – the method creates a list of FlightPath objects. It takes the list of moves by the drone as an input and parses it to the flightpath object
- createDeliveriesDatabase() – the method creates the deliveries table in the database. If the table already exists then it drops the table and creates a new table. It then adds in the delivery objects to the table
- createFlightPathDatabase(List<LongLat> moves) – the method creates the fightpath table in the database. If the table already exists, then it drops that table and creates a new table. it then populates this new table. It takes the list of moves by the drone as an input

GeoJsonParser.Java

The class is used for reading the geojson files in the webserver and also for writing geojson files. The class has the following variables:

- noFlyZoneFeatures – which is a list of features. Each feature represents a polygon. The polygons make up the no fly zone
- landmarkFeatures – which a list of features. Each feature is a point on the map which are landmarks on the map
- client – this is the HTTP Client used for accessing the webserver
- urlStringNoFlyZones – the url string which points to the location of the no fly zones geo json file in the webserver
- urlStringLandmarks - the url string which points to the location of the landmarks geo json file in the webserver

The class has the following methods which are all public and static as they are all called from different classes :

- getNoFlyZones() – this method saves the no fly zone features in the noFlyZoneFeatures variable
- getLandmarks() - this method saves the landmark features in the landmarkFeatures variable
- movesToFCCollection() – given a list of moves made by the drone, the method converts it to a list of points and then a geometry object and finally a feature collection. It then returns the json string of the feature collection.
- outputGeoJsonFile -given a json string as input, this method creates a new geojson file in the current working directory and then writes the string to the file

LongLat.Java

The class is used to represent a location on the map, given its latitude and longitude value. It also has a couple of methods that can be used on a Longlat object. The variables in the class are:

- longitude – the double value of the longitude of the current object

- latitude – the double value of the latitude of the current object
- point – the geojson point value of the current object
- oneMove – the value of one move that the drone can make in degrees
- confAreaRight – the longitude value of the right-most point of the confinement area
- confAreaLeft – the longitude value of the left-most point of the confinement area
- confAreaTop – the latitude value of the top-most point of the confinement area
- confAreaBottom – the latitude value of the bottom-most point of the confinement area
- Appleton – the LongLat object that represents the coordinates of Appleton tower.

The class has the following classes which are all public:

- isConfined() – checks if the object is inside the confinement area
- distanceTo() – returns the double value of the eucleadian distance between two LongLat objects.
- closeTo()- checks if 2 objects are close to each other.(having a distance of less than or equal to 0.00015)
- inNoFlyZone() – checks if a move by the drone to the next position puts it in the no fly zone by the line-line intersection method
- inNoFlyPolygon()- checks if a LongLat object is in a no fly zone polygon by using the point in Polygon method
- angleDirectionTo() – returns the integer value of the angle between two objects

Orders.Java

The class is used to get and store the order details to be accessed by other classes later on in the program. The class has a number of different variables which are:

- pickUpCoordinates - hashmap that maps an order number string to a array of coordinates stored in a list of doubles. The coordinates are the coordinates of the shops that the drone has to pick up the orders from
- pickUpWords – a hashmap that maps an orders order number to the WhatThreeWords values of the shops it need to pick the food from
- deliveryCoordinates - a hashmap that maps an order's order number to the coordinates of the delivery location which is stored as a double array
- deliveryAddress – a hashmap that maps an orders order number to the WhatThreeWords values of the delivery location
- items – a hashmap that maps an orders order number to the list of item names ordered in that order
- orderNos - a list of order numbers stored as string for the specific day
- deliveryDate –the date that was specified in the input arguments
- jdbcString –The JDBC String Used to access the database. The Database Port is received as an input when the program is ran
- client -the HttpClient that is shared between all HttpRequest

The class has the following methods:

- getCoordinatesAndOrders() – the method is used to run all of the private methods in this class
- getOrders() – the method is used to fetch and store the order numbers and delivery addresses for a certain date from the database
- getOrderDetails() – the method is used to fetch the items ordered in a certain order from the order details database
- getDeliveryCoordinates() -given the WhatThreeWords string of a delivery location, the method searches the webserver for its details.json file to get the coordinates associated with that location
- getPickUpWords() – given the items ordered in a certain order, the method searches through the menus.json file in the webserver to find the address of the shop that sells that item
- getPickUpCoordinates() - given the WhatThreeWords string of a pick up location, the method searches the webserver for its details.json file to get the coordinates associated with that location

HexGraph.Java

The class is used for creating a graph for the drone to traverse and also for implementing the TSP algorithms which will sort the orders based on which order to complete first. The class has the following variables:

- OrdersCompleted - the list of order numbers of orders that the drone was able to complete for the day
- HoverLocation - a list of integers that represent the index of the movements in the drones movements when the drone was hovering
- OrderNumberChanges - a hashmap that maps an integer index to the order number string. the integer represents the index in the movement of the drone when the drone has moved on to the next order
- heightOfTriangle - the height of the triangles created in the graph.

The class has the following methods for building the graph:

- shiftedRow() – check if the row should be shifted to the right or not
- genNodesOnHexGraph() – the method plots nodes (vertices) on the graph by storing their locations in a 2d list. The nodes are one more(0.00015) apart from each other and always within the confinement area
- genPathOnHexGraoh() – this method generates the nodeEdges on the graph. It takes the node previously found and plots edges between neighbour nodes. This forms equilateral triangles between 2 neighbour nodes and for each nodes a heaxagonal shape appears to surround it.
- genDroneHexGraph() – this method plots the nodes and edges on a new graph and returns it.
- genValidHexGraph() – this method checks every node and edge to make sure that it is in the confinement area and not in the no fly zone. If not, the node or edge is removed.

- getDeliveryNodes() – given the list of nodes in the graph, the method find the nodes close to a delivery location
- getPickUpNodes()-given the list of nodes in the graph, the method find the nodes close to a pickUp location
- getAppleton() - given the list of nodes in the graph, the method find the nodes close to a Appleton Tower.

2. Drone Control Algorithm

The algorithm works such that the route is calculated and its movements are returned. The algorithm that calculates the shortest path is simple but the graph that allows this algorithm to work is what can be confusing

2.1 Graph

I had initially wanted to use the Christofides algorithm as a TSP algorithm to solve the problem of which order to complete first. For that algorithm to work, I would need to create a graph of equilateral triangles whereby for each node, it would have at least 2 neighbouring node which will form an equilateral triangle.

To build the graph I first had to find the locations of each node. Hence starting in the top left corner of the confinement area and working down to the bottom right corner, I would more one more (0.00015 degrees) to the right and 1 height down(calculated using Pythagoras theorem). This would allow me to find the position of each node in every column and every row that is in the confinement area

Then I connected each node with its neighbouring nodes with nodeEdge objects. Each node would have a minimum of 2 neighbour nodes and a maximum of nodes. Hence for most nodes, it appears to have a hexagon shape surrounding it. This is because for the triangles to be equilateral, the angle of each corner would have to be 60 degrees hence there would be a maximum of 6 nodes.this means that from a given node, the drone can move in 6 different directions

I then added all of the nodes and edges to a graph. I then checked if each node and edge on the graph was valid. A node or edge is valid if it is in the confinement area,not in the no fly zone and does not cross the no fly zone. If a node or edge is invalid, then I would remove the node or edge entirely. Since there are no nodes or edges in the no fly zone or outside the confinement area, the drone cannot traverse there,hence it is impossible to be in an invalid location.

2.2 Traversal

To traverse the graph the drone first had to find the nodes close to the locations it wants to go to. This is because the locations of interest may not exactly be on a node’s location. However since the distance between each node is 0.00015 degrees and the drone is allowed to pick up or deliver an order if it is close to the locations (having a distance of less than 0.00015 degrees) there will always be a node close to each location of interest.

Once the nodes are found, the drone can find the shortest path. It uses one of the TSP algorithms to find the most optimum order of the orders to complete. The drone uses Dijkstra shortest path algorithm from jgrapht to find the shortest path on the graph I had previously between 2 points. I had tried using Astar shortest path algorithm with the eucleadian distance as the admissible heuristic but it did not work any much better than Dijkstra so I stuck with Dijkstra (you can use the A* algorithm by uncommenting it in the code and commenting the Dijkstra line). It first gets the route from Appleton to the first pick up location and stores the nodes it visited on a list. If there is another location to pick up the order from, it moves there otherwise it moves to the delivery location and stores those nodes in the list. We assume that the drone should be able to complete at least one order before it runs out of battery. It then repeats the process where it moves from the current order's delivery location to the next order's pick-up location. However, it first checks if the drone has enough moves left to complete the order and make its way back to Appleton tower. If it does not, it skips that order and moves to the next order until it can find an order it can complete. If it cannot complete any of those orders with the number of moves left. It just returns to Appleton tower

While the drone is moving between orders, the index of the nodes in the list of nodes when the drone hovers and moves on to the next order is recorded. This will be used later when writing the data to the database

The algorithm is simple as for each order I just need to find the path with the shortest distance between the pick up and delivery locations and add it to the list of movement. I do not need to worry about the drone moving into the no-fly zone or out of the confinement area as I know that would never happen since I have already removed those nodes and edges.

2.3 the TSP Algorithms

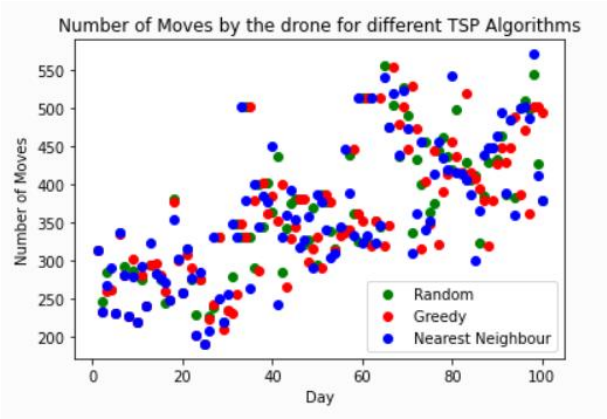
To maximise the amount of money earned per day I would need to use an algorithm that would prioritise certain orders over the other. I had initially wanted to use Christofides algorithm as it seemed to have the best results. However, due to a lack of time I was unable to implement it. Instead I implemented 3 others which are the greedy approach,nearest neighbour approach and lastly the random approach.

The random approach would sort the list of order numbers randomly. This approach worked fine when the number of orders per day was low like 5 or 6 but got worse as the numbers increased.

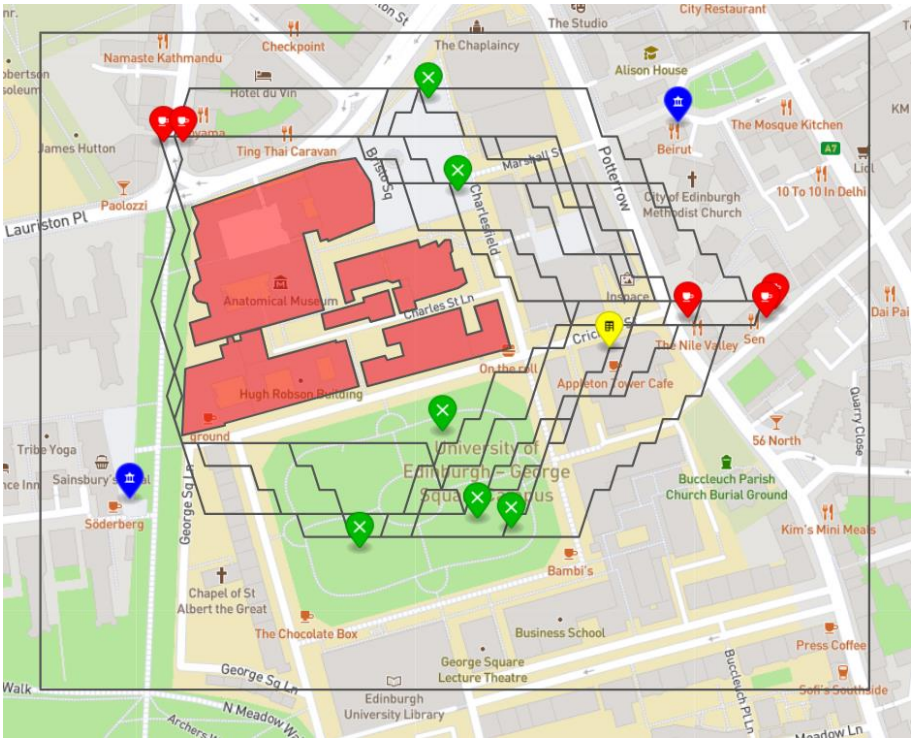
The next approach I tried was the nearest neighbour approach where given the location of the drone, the algorithm would choose the order which has the nearest pick-up location to its current location and choose that order next. This order worked well in most cases, but it prioritised getting the lowest number of moves over getting the most money hence on days when it was impossible to complete all orders it made less money than the greedy approach

The greedy approach seemed to be the best among the three as it prioritised the money earned which is the main objective of the system. Even on the day with the greatest number of orders which is 27/12/2023 the percentage of monetary value is 84% which is higher than the other algorithms. Hence, the greedy approach gave the most optimal order or orders that the drone should complete. I also tested each approach on 100 different days and plotted a scatter

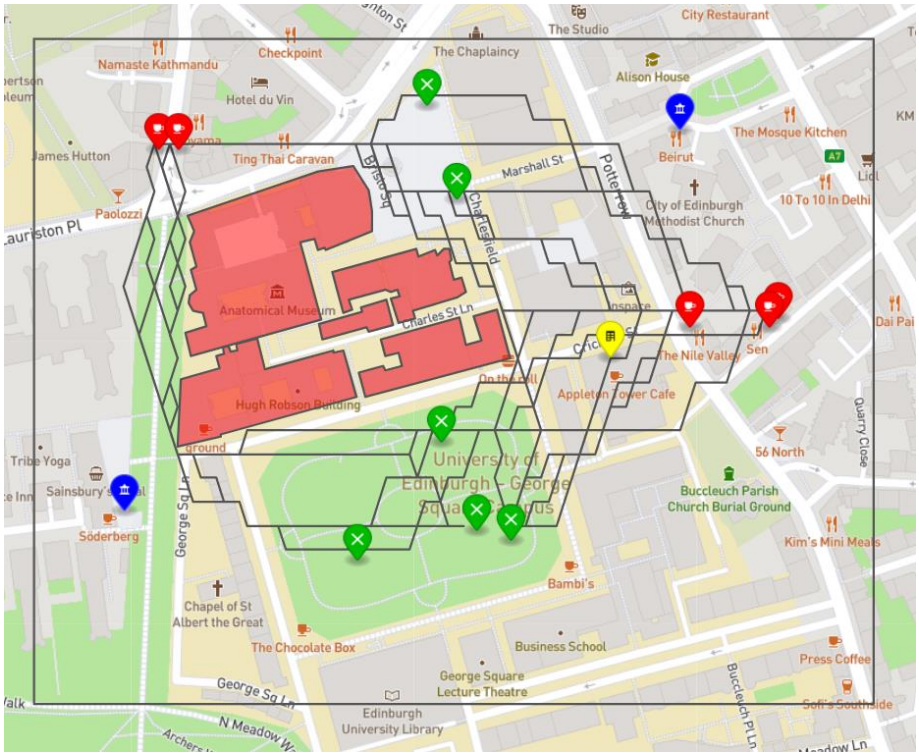
diagram to convey my results. From the diagram it is obvious that the greedy approach and nearest neighbour approach have similar results.



2.4 Figures



Drone path on 11/11/2022



Drone path on 12/12 /2022