Informatics Large Practical

Drone Delivery Report – Coursework 2

Sergio Miguez Aparicio – s1960766

The University of Edinburgh – *December 2021*

Table of Contents

Software Architecture Description *2*

General Overview *2*

Classes Documentation *3*

- *App Class* *3*
- *Basic Utils Class* *3*
- *Database Handler Class* *3*
- *Drone Class* *4*
  - *Drone Movement Class* *4*
  - *Drone Control Algorithm Class* *5*
- *GeoJSON Handler Class* *6*
- *HTTP Client Class* *6*
- *Line_ Class* *6*
- *LongLat Class* *6*
- *Menus Class* *6*
- *Restaurants Class* *7*
- *Words Handler Class* *7*
- *Interface - Request Utils* *7*

Drone Control Algorithm *7*

- *Main Aim, Movement Restrictions, Confined Area* 7
- *Sorting Orders* *8*
- *Order Selection* *8*
- *Flight Simulation Heuristic* *8*
- *Restaurants Heuristic* *8*
- *Finding Landmarks Heuristic* *8*
- *Avoiding No Fly Zones, Out of Confined Area & Loops* 9
- *Returning to Base* *9*
- *Writing a GeoJSON File* *9*
- *Drone Flight Figures* *9*
- *Generating Submission Files* *10*
- *Results and Statistics* *10*

References *10*

Software Architecture Description

The application is formed by 11 classes which are basic to perform the operations of this project. These are: *App*, *Basic Utils*, *Database Handler*, *Drone*, *GeoJSON Handler*, *HTTP Client*, *Line_*, *LongLat*, *Menus*, *Restaurants* and *Words Handler*. Adding on, an interface, *Request Utils* has also been used.

The user must input the following values to initialise correctly the program:

- a date of year 2022 or 2023 (the deliveries simulation will run for this specific date).
- a port number to a web server (which will contain the information of no-fly zones, landmarks, the menus of all the restaurants, and What3Words details to get coordinates).
- a port to a database (which contains a list of orders, their details, and where the flight path information and completed deliveries must be written).

General Overview

The main structure can be found in **App**, where the general pipeline of the project is developed. It is where the user inputs are read and processed into the program. This class creates 5 key Objects which are needed to make the deliveries. These are a *Drone*, *Database Handler*, a *Words Handler*, a *GeoJSON Handler*, and a *Menus* Object. They are basic and will retrieve the required information to do the deliveries, the no-fly zones, landmark coordinates, all the menus and their items; and the drone which will effectuate the deliveries. Having all this information, the program (specifically the Drone), is ready to start taking decisions of which, through where and how, the deliveries should be made.

The **Database Handler** is the class through which any type of interaction with the database must be done. When an instance of *Database Handler* is created, it connects to a derby database, sets up two tables: "*DELIVERIES*" and "*FLIGHTPATH*"; and retrieves the orders information of the date input by the user. It generates two important HashMaps "*hashmap_orderNoToW3w*" and "*hashmap_orderNumbToItems*" key to get pieces of information from the orders in a fast way. It also generates s list of all the orders numbers of a day. The class has methods implemented to create and drop tables, get the orders of a specific date, insert values, retrieve, and print all the information.

The **Drone** contains all the functions which involve taking decision of which deliveries must be done, the path it should take, and the logic used to avoid no-fly zones or getting out of the confinement region. The class contains two subclasses, **Drone Movement** (which will contain all the methods that are related to the displacement of the drone); and **Drone Control Algorithm** (dedicated to the decision making of the drone to effectuate deliveries, simulate paths and checking that the flying restrictions are being obeyed).

The abstract **Basic Utils** class contains the variables and constants used across multiple classes. This would be the input data given by the user, and a constant with the hovering angle value. Adding on, it has the method to print and save the final statistics.

The **GeoJSON Handler** has the main task of generating an output file which will contain the flight path of the drone of that date. Adding on, it is used to read GeoJSON files from the web server which contains the information of the no-fly zones, and the coordinates of the landmarks. These are all parsed into Objects the *Drone* can work with to take decisions.

All the requests to the web server are done through the class **HTTP Client**. It contains the static *CLIENT* attribute and the method used in all operations "*make request()*". These are all called from the *Request Utils Interface* used by all the classes which connect to the web server (*GeoJSON Handler*, *Menus* & *Words Handler*).

The **Line_** class is used to define 2D line Objects by knowing two *LongLat* points that are contained in the line. This Object is used in multiple classes (*Drone* & *GeoJSON Handler*) to describe limits of no-fly zones, the path the drone is doing, or the direction the drone wants to move at.

All the coordinates of the project are in form **LongLat** Objects. This class contain basic functionalities used across the classes *Drone*, *GeoJSON Handler* & *Database Handler*. From checking distances to a new coordinate having given the angle to make the move. When an instance of *LongLat* is created, the given values for the geographical longitude (*lng*) and latitude (*lat*) are checked to ensure the point is valid.

The **Menus** class is used to request to the web server and parse all the information of all the restaurants that can be part of an order, the items each one sells and their prices. The web server returns this information in a JSON format. When an instance of *Menus* is created, it will automatically populate the two HashMaps, "*hashMap_Item_Price*" and "hasMap_Item_w3w", for each restaurant with the information acquired. Another key method of the class is "*getDeliveryCost()*" used to sort the deliveries in *App*.

The **Restaurants** class is used to parse the information retrieved in *Menus* class. A Restaurants Object will contain all the information of a restaurant. The key properties are stored in two HashMaps, "*hashMap_Item_Price*" and "*hashMap_Item_w3w*", which are filled when called in the *Menus* constructor.

The **Words Handler** class contains all the tools required to request to the web server the coordinates used through the program in What3Words format. When an instance of *Words Handler* is constructed, the request to a list of locations is performed using the method "*requestCoordinate()*". In this case, a list with the w3ws of the delivery locations is passed into the constructor. Each coordinate is added into an important HashMap used across the project named "*hashMap_w3w_coordinates*". When ever a

coordinate is needed, the HashMap is checked and if it is not found, a new request to the web server is done and the new information is added into the HashMap. With this approach, the program will only run with the information really required to process the orders of that specific date, optimising the runtimes and the program.

The **Request Utils Interface** is used to define the constants used across all the classes that make requests into the web server (*GeoJSON Handler*, *Menus* & *Words Handler*). Adding on, it includes the abstract method to make a JSON request of any type. This also includes GeoJSON files.

Through all these classes, the structure of the program is fully defined, and all the functionalities required in the specifications are completely covered. This distribution ensures the code to be readable and easy to maintain. The *App* class will contain the general process and the *Drone* will contain the algorithm and the methods to move through the map. Moreover, there classes which will have individual specific aims (*Database Handling*, *Words Handling*, *Menus*, *HTTP Client* & *Request Utils*), or a more abstract usage (*LongLat*, *Line_*, *Restaurants* & *Basic Utils*). Adding on, the methods inside the classes form atomic pieces of logic which have high relevance in their class and are used to build up to perform complex decision and calculation processes, leading to high cohesion. Another important aspect to consider is encapsulation. All the possible attributes and methods used across the program are private always, except especial cases where they had to be turned into package protected. The *Drone* will correctly encapsulate all the possible methods, letting *App* just access the sorting restaurants method, requesting making all the deliveries and a returning to appleton. With all these characteristics, the project ensures a correct behaviour, readability, maintenance, and independence /encapsulation of information.

Class Documentation

App Class

Attributes

- Database Handler *db* — protected - instance to request deliveries and write flight path.
- Words Handler *w3ws* — protected - instance to get and request coordinates.
- GeoJSON Handler *geo* — protected - instance to get, parse and writes GeoJSON files.
- Menus *menus* — protected - instance to get the menus of restaurants.
- Drone *drone* — private - flying drone, contains all the functionalities needed.
- ArrayList<String> *orderNumbSort* — protected - all order numbers of a day sorted.

Methods

- ***main():*** (public) is the main method of the class and the project. It contains the main logic structure of the program. It gets the user inputs into the **Basic Utils** class, create instances of the **Database Handler**, **Words Handler**, **GeoJSON Handler**, and **Menus** classes. Through all these, the program will request to the database and the web server just the initial information required to start processing the information and taking decisions for the orders of the specific date given by the user. If more information is required, it will be retrieved progressively. The orders are sorted by decreasing cost of order using the method "*sort orders()*" and the deliveries are done through "*make deliveries()*". Finally, the drone returns to the base using "*return to appleton()*", the GeoJSON output is written, and the results/statistics of the day are shown to the user.
- ***variable init(String[] args):*** (private) used to store the inputs of the user into the variables of **Basic Utils** class and checks that they are valid. It also initialises the other general variables of the **App** class. It takes the string array *args* to save the user's inputs.

Basic Utils Class

Attributes

- String *day* — protected - the user input date set into the program.
- String *month* — protected - the user input month set into the program.
- String *year* — protected - the user input year set into the program.
- String *port web server* — protected - the user input of the web server port to connect to.
- String *port database* — protected - the user input of the database port to connect to.
- int *HOVERING ANGLE = -999* — protected - constant with the special hovering angle value.

Methods

- ***print results():*** (protected) outputs the general statistics of the deliveries date. The number of completed orders out of the total, the number of movements done, and the total cash made. It also creates a txt file "drone-DD-MM-YYY-results.txt" with results.

Database Handler Class

Attributes

- Connection *connection* — private - stablish the connection to the database.
- Statement *statement* — private - creates SQL statements and make requests to the database.

- DatabaseMetaData *databaseMetaData* private - to check if a table exists.
- PreparedStatement *preparedStatement* private - prepares SQL statements.
- String *done deliveries = "DELIVERIES"* private - constant of the name of completed deliveries table.
- String *done flightpath = "FLIGHTPATH"* private - constant of the name of flightpath database table.
- ArrayList<String> *orderNumbList* private - list of the order numbers of a date.
- ArrayList<String> *ordersW3WLocationList* private - list of w3w of the delivery points of a date.
- HashMap<String, String> *hashmap_orderNumbToW3w* - private - get access to w3w location having an orderNumb.
- HashMap<String, ArrayList<String>> *hashmap orderNumbToItems* - private - get access to items of an order.
- LongLat *previous toLongLa* private - used to check if there are any points missing.
- boolean *first move* private - indicates if it is the first movement being added.
- int *completed deliveries* private - counter of the completed deliveries.

Methods

- ***update metadata():*** (private) updates the metadata of the database.
- **does table already exist(String table name):** (private) calls the "update metadata()" method and returns a boolean.
- ***drop table(String table name):*** (private) if the table exists, it is eliminated from the database.
- ***create table deliveries():*** (private) if the table "DELIVERIES" existed, it is dropped, and then a new one is created.
- ***create table flightpath():*** (private) if the table "FLIGHTPATH" existed, it is dropped, and then a new one is created.
- ***get orders of day():*** (private) retrieves the information from the ORDERS & ORDERDETAILS tables of the database. This gives all details to perform the orders of a date.
- ***Insert completed delivery(String orderNumb, String deliveredTo, int costInPence):*** (protected) the method is prepared to insert new information into the DELIVERIES table, when a delivery has been completed.
- Insert flight paths(String orderNumb, Double fromLongitude, Double fromLatitude, Integer angle, Double toLongitude, Double toLatitude): (protected) inserts a movement of the drone's flightpath into the FLIGHTPATH table of the database.
- ***Check gaps(double lng, double lat):*** (private) used to prevent writing values into the database if there are missing flying path coordinates. To operate error checking.
- ***get Completed deliveries():*** (protected) returns the total amount of completed deliveries.
- ***get orderNumb List():*** (protected) returns the list of order numbers.
- **ArrayList<String> *get_OrdersW3WLocationList():*** (private) returns the list of w3w location used in a date.
- HashMap<String, ArrayList<String>> *getHashmap_orderNumbToItems():* (protected) returns HashMap orderNumb-Item.
- HashMap<String, String> *get_Hashmap_orderNumbTow3w():* (protected) returns HashMap orderNumb-W3W.

Drone Class

Attributes

- Drone Movement *drone movement* protected - gives the drone the capacity to move.
- Drone Control Algorithm *drone control algorithm* protected – gives the drone the capacity to make decisions.
- int *movement counter* private - number of movements done.
- int *cash made* private - total amount of money done till the moment.
- LongLat *drone position* private - stores the coordinates where the drone is.
- String *orderNumb* private - order number of the order being processed.
- ArrayList<LongLat> *return path* private - coordinates to return to base.
- int *MAX MOVEMENTS = 1500* private - constant with the total battery.

Methods

- ***get movement counter():*** (protected) returns the total number of movements the drone has done.
- ***get drone control algorithm():*** (protected) returns the drone's capacity of taking decisions (to make orders, to order restaurants, check that the flying restrictions are being obeyed).

Drone Movement Class

Attributes – No attributes

Methods

- ***go to(LongLat to place, ArrayList<LongLat> path):*** (private) method moves the drone the necessary times till it reaches the *to place* coordinate using the "*closeTo()*" method. For each step, the "*make a move to()*" method will be called with the drone position and the new coordinate is recorded in *path.*
- ***make a move to(LongLat origin, LongLat place):*** (private) returns the new coordinate LongLat after performing a move. It calls the "*get angle()*" and "*nextPosition()*" methods.

- ***hover move(LongLat position):*** (private) adds a hover move into the database and to the GeoJSON file. Calls the method "*insert flight paths()*" of the *Database Handler* instance.
- ***get angle(LongLat position, LongLat place):*** (private) estimates the optimum angle the drone should do to get closer to *place*, the new angle is checked using the "*good direction()*" method and if it is not a valid angle, the method will test the angles +/- 120º degrees from the suggested till it finds a valid option. If no other valid angle is found, the drone will use the initial angle.

Drone Control Algorithm

Attributes

- ArrayList<Restaurants> *restaurants* — private - contain the restaurants that are part of a delivery.
- ArrayList<LongLat> *path* — private - coordinates that describe the drone's displacement
- LongLat *base* — private - coordinates of Appleton Tower.
- LongLat *destination* — protected - coordinates of place to reach.
- LongLat *hover* — protected - special LongLat to identify hovering move.
- double *APPLETON LONGITUDE* — protected - constant Appleton longitude's coordinate value.
- double *APPLETON LATITUDE* — protected - constant Appleton latitude's coordinate value.

Methods

- ***sort orders():*** (protected) is the function where the order numbers are sorted in decreasing value of the cost delivery.
- ***make deliveries():*** (protected) makes all the possible deliveries. It iterates through each order number of the orders list (already sorted) and calls the method "*make a delivery()*" which will try executing the delivery if possible. If it cannot be performed, the program will skip to the next delivery till there are no more orders to complete.
- ***make a delivery():*** (private) it is where the program decides if a delivery should be done or not. Firstly, the method will get the destination information and the list of restaurants involved in the order. If any restaurants coordinates are missing in the HashMap "*get HashMap w3w coordinates*" and "*get HashMap orderNumbToW3w*" it will be requested to the web server. The program simulates the full path that the drone would do to reach the restaurants and delivery location using the method "*sort restaurants()*" and then simulates the full path to return from the destination to Appleton Tower using "*get path to appleton()*". If the number of movements in total of both paths is less than the remaining movements/battery value, the delivery will happen and the method "*realise simulated path()*" is called.
- ***sort restaurants():*** (private) simulates all the possible paths to reach a destination going through the landmarks and the restaurants needed. In the case that the drone must go to two restaurants A & B, split paths are calculated going from the drone position first to A, then B, and then destination; and going from drone position to B, then A, and then destination. Both paths are stored int temporal ArrayLists of LongLats and the one which take less drone movements will be saved into the *path* ArrayList. Must be remarked that this is a simulation, and the drone has not been moved yet.
- ***return to appleton():*** (protected) simulates the path to get back calling "*get path to appleton()*" and then writes the path found into the database and the GeoJSON file.
- ***get path to appleton():*** (private) simulate the returning path from the destination, back to Appleton Tower. Similar to *sort restaurants* method, the path is written into the *return path* ArrayList, and the drone has not been moved from its position.
- ***realise simulated path():*** (private) method call when a delivery has been approved and the delivery must be completed. It iterates through each coordinate adding into the "FLIGHTPATH" database table and the GeoJSON file the positions of the drone. Finally inserts the completed delivery information into the "DELIVERIES" table and records the total cash made.
- ***get restaurants of order(String item)****:* (private) searches the item among the restaurants of the database using the HashMap "*get HashMap Item w3w*". Used to get the restaurants that must be visited in the delivery process.
- ***get path delivery(ArrayList<Restaurants> restaurants, ArrayList<LongLat> path):*** (private) Operates the full process of estimating the movements of a delivery given the restaurants to visit and an ArrayList *path* to record the movements. For each restaurant it calls the methods "*get path through landmarks()*" to get the movements in case a landmark must be used; and "*go to()*" to perform the movements from the position of the drone (starting position or continue from landmark) to the restaurant. After this, the same process is followed to reach the destination.
- ***get path through landmarks():*** (private) checks if the direction the drone is trying to perform has any no-fly zone obstacles. If there are the method "*get landmark()*" is called and then the movements to the landmark are simulated through "*go to()*" method.
- ***get landmark(LongLat position, LongLat destination):*** (private) gets the landmark from which the *destination* coordinate can be reached and has a clear path from the given *position* (go through it without going into no-fly zones) and that will involve the least number of steps (optimising distance). To check the path, the method "*good direction()*" is used.
- ***good direction(Line_ direction):*** (private) checks if the line given intersects with any of the no-fly zone limits or is out of the confined area. To perform this task, the method "*find intersection()*" is called.
- ***find intersection(Line_ direction, Line_ no-fly zone):*** (private) method based in the theory of the article "*How to check if two given line segments intersect?*" of GeeksforGeeks. It finds the intersection between two lines/segments (limited by their

definition points). Check the orientation of the points of the lines using the "*check orientation()*" method and if the intersection is inside the segment "*on segment()*".

- ***check orientation(LongLat a, LongLat b, LongLat c):*** (private) checks the geometric orientation between three points. These could be collinear, clockwise or counterclockwise. This method is based in the theory of the article "*How to check if two given line segments intersect?*" of GeeksforGeeks.
- ***on segment(LongLat a, LongLat b, LongLat c):*** (private) checks if a point *b* is part of the segment defined by *a* and *c*. This method is based in the theory of the article "*How to check if two given line segments intersect?*" of GeeksforGeeks.

GeoJSON Handler Class

Attributes

- FeatureCollection *output file* — private - GeoJSON FeatureCollection to be written into the output file
- List<Point> *flights* — protected - list of Point locations which describe the drone flightpath.
- ArrayList<Feature> *features* — private - list of features to be written into the GeoJSON.
- ArrayList<LongLat> *landmark points* — protected - list of landmark coordinates.
- ArrayList<Line_> *no fly zone limits* — protected - list of all the Line_ s which delimit the no-fly zone areas.

Methods

- ***write GeoJSON():*** (protected) writes and stores the fly path of the drone into a GeoJSON file named: "drone-DD-MM-YYYY.geojson".
- ***reading GeoJSON landmarks():*** (private) retrieves from the web server the Points of the landmarks, cast them into LongLat Objects and adds them into the *landmark points* list.
- ***reading GeoJSON no fly zone():*** (private) retrieves from the web server the Polygons which delimit the no-fly zones and cast them into Line_.
- ***from Point to LongLat(Point a):*** (protected) used transform a Point into a LongLat with same coordinates.
- from Polygon to lines(ArrayList<Line_> list, Polygon polygon): (protected) casts a Polygon into a List of Line_.

HTTP Client Class

Attributes

- static HttpClient *Client* — private - used to connect to the web server and make requests.
- int *CONNECTION ERROR = 404* — private - constant shown if any error occurs.

Methods

- **HttpResponse<String>** ***make request(String url):*** method to perform any type of request to the web server.

Line Class

Attributes

- LongLat *pointA* — protected - point contained in the line, defines beginning of the segment.
- LongLat *pointB* — protected - point contained in the line, defines ending of the segment.

Methods No methods are contained in this class.

LongLat Class

Attributes

- double *lng* — protected - longitude value of a coordinate.
- double *lat* — protected - latitude value of a coordinate.
- double *LENGTH DRONE MOVEMENT = 0.00015* — protected - constant of the distance the drone moves each time.
- double *TOP MEADOWS LONGITUDE* — protected - constant of the longitude of Top Meadow's.
- double *TOP MEADOWS LATITUDE* — protected - constant of the latitude of Top Meadow's.
- double *KFC LONGITUDE* — protected - constant of the longitude of KFC.
- double *KFC LATITUDE* — protected - constant of the latitude of KFC.

Methods

- ***isConfine():*** (protected) checks if the point is strictly in the drone confined flying area, delimited by the square Top Meadow's and KFC points.
- ***distanceTo(LongLat second reference):*** (protected) calculates distance between the actual coordinate and a new one.
- ***closeTo(LongLat second reference):*** (protected) checks if a coordinate is strictly below a distance unit of 0.00015.
- ***nextPosition(int angle):*** (protected) calculates a new coordinate given the angle the drone wants to move to.

Menus Class

Attributes

- int *DELIVERY CHARGE = 50* protected - constant base charge in pence to perform a delivery.
- ArrayList<Restaurants> *restaurants* protected - list of the restaurants and all their information.

Methods

- ***updateMenus():*** (private) retrieves the menus information from the web server, and parts it into the restaurants list.
- **getDeliveryCost(ArrayList<String> items):** (protected) gets the cost of a delivery using the "HashMap Item Price".

Restaurants Class

Attributes

- String *name* private - name of the restaurant.
- String *location* private - location in w3w of the restaurant.
- ArrayList<Menus_db> *menu* private - contains the menu of the restaurant in a list from.
- HashMap<String, Integer> hashMap_Item_Price protected - get prices of items in a fast way.
- *HashMap<String, Integer> hashMap_Item_w3w* protected - get the locations of items in a fast way.

Methods

- ***createHashMap():*** (protected) builds the HashMaps after getting the response of the web server and parsing it.
- ***getLocation():*** (protected) return the location in w3w of a restaurant.

Inside the **Restaurants** class, the **private static class Menus_db** can be found. This is only used to parse the information of the menus returned from the web server,

Attributes (private static class Menus db)

- String *item* private - the name of the item in the menu.
- String *pence* private - the value in pence of the item.

Methods No methods are contained in this class.

Words Handler Class

Attributes

- String *url* private - URL of the request.
- W3W *details* private - stores the w3w JSON answer of the server.
- HashMap<String, LongLat> hashMap w3w coordinates private - get coordinates having w3w locations.

Methods

- ***requestCoordinate():*** (protected) check if the coordinate is in the HashMap, and if it is not, its requested and added.
- **get_HashMap_w3w_coordinates():** (protected) returns the "hashMap w3w coordinates".

Inside the **Words Handler** class, the **private static classes W3W** and **Square** can be found. These are only used to parse the information of the What3Words coordinates that are returned from the web server.

Attributes (private static class W3W)

- String *words* private - contains the coordinates in w3w format.
- LongLat coordinates private - contains the coordinates in LongLat object.

Request Utils Interface

Attributes

- String *HTTP = "http://"* constant with the starting string of any request to the web server.
- String *machine name = "localhost"* constant name of the machine to which the requests will be done.
- String *jdbc protocol = "jdbc:derby//"* constant with the starting string of any request to the database.

Methods

- ***requestJSON(String url):*** (default) makes a request to the web server. This function is shared among all the classes which make web server requests and is prepared to receive any type of JSON/GeoJSON response.

Drone Control Algorithm

Main Aim, Movement Restrictions and Confined Area

The program must optimise the path to complete as many deliveries as possible trying to achieve the highest monetary value delivered. This is a clear example/application of the Travel Salesman Problem as the order of deliveries and the way they are processed must be done trying to minimise the flight path of the drone. All these will take place in a limited space (Square) delimited by 4 restaurants coordinates of the city of Edinburgh. The drone must remain at all time in this area. There will be areas/building the drone must not fly through called no-fly zones. These will be restricted areas or obstacles the drone must avoid while completing the deliveries. The system to avoid getting out of the confined area or into no-fly zone areas is explained at the entry **Avoid No-Fly Zone Areas, Out of Confined Area & Loops**. The drone has a battery which allows a total of 1500 fly movements and algorithm

must ensure that there will be enough battery to return the base at the end of the day. This restriction must be considered as some deliveries might not take place as there is not enough battery to accomplish them or to return afterwards to Appleton Tower. Furthermore, the movements are restricted to have a distance of 0.00015 distance units and the angles to get new positions will be in the range [0, 350]. Only those angles divisible by 10 are valid. In conclusion, the program must ensure the correct movement of the drone through the space, avoiding the no-fly zones or getting out of the confined area, and try to achieve the highest monetary value delivered of that date.

Sorting Orders

The method of evaluating the drone's performance is through the *percentage monetary value* delivered of the date. A greedy algorithm was used to try optimising the selection of the orders to deliver. The program retrieves the information necessary to calculate the cost of each delivery and sorts them in decreasing order by the money to be gained in the delivery. This process does not ensure the best combination of orders, but in most cases, it will perform correctly considering the small needed computational effort to do this task.

Order Selection

When an order is taking place, the drone will simulate the path that it will take. This process allows calculating the cost of battery movements that the delivery will have if the flight is completed. These allows the drone to know if the delivery can be safely completed or not. The drone will run two simulations in this process:

Flight path to the destination and Returning from the destination to Appleton.

In this way it can be checked if the remaining battery is enough to make the full delivery and return to the base. In the case it cannot, the drone will simulate its way back to Appleton and then complete the flight.

Flight Simulation Heuristic

Whenever a flight is being simulated the following algorithm will take place to go through the landmarks if needed. If no landmarks are needed the method *go to()* will be used (Also described in the following pseudo code) to reach the final position:

```
get path through landmarks(to place, path) =>
        good direction(drone position, location) => check if no intersections with
              no-fly zones are found between the drone position and the location.
              landmark = get landmark(drone position, location) => In case there
                    is, check all the landmarks reachable from the drone's position
                    and that have a clear path to the location. Return the one which
                    implies the least number of steps.
              go to(landmark, path) => while the drone is not close to landmark,
                    make a move and store the information in path.

go to(to place, path) =>
      while drone position is not close to place
            drone position = make a move(drone position, to place) => gives a new
                  coordinate by calculating the optimum angle
```

This process is used to reach a destination point, going through landmarks if needed. It is also used to get back to Appleton. The key aspect of this implementation is that the path is saved. Then, the flight is checked to ensured that it can be completed. Finally, the ending process would just be including each of the entries of the path into the database and the GeoJSON file. The algorithm is simulations all the possible paths that can be created by going through each landmark, despite adding computational cost, this gives the benefit of choosing the shortest flight, avoiding wasting movements in unnecessary longer paths.

Restaurants Heuristic

When a simulation of a delivery flight is taking place, two different cases can occur. The delivery implies going through one, or two restaurants. In the first case, there path is immediately traced, but in the case of two restaurants, the drone could take two different paths. From Origin to Restaurant 1, to Restaurant 2, to Destination; or from Origin to Restaurant 2, to Restaurant 1, to Destination.

The key aspect of the method is that, when having multiple restaurants to fly through, the drone can compare and decide in which order it should fly to the restaurants; and complete the delivery in the most optimum number of steps.

Finding Landmarks Heuristic

In order to get correctly to whichever location, the system will try performing a simulated path to the destination directly. Not always this path is possible and intersection with a no-fly zones will take place. To solve this issue, the system will try searching for a landmark to which the drone will have a clear flight and, at the same time, from which the destination can be reached. The method *get landmark()* will go each landmark and compare which of the valid landmarks generates the shortest distance. In the case no landmarks are found to reach the destination, the system will just try going through the optimum path and if it crashes with a no-fly zone, it will try adapting the movements angles to avoid it using the methods described in the **Avoiding No-Fly Zones** entry.

Avoiding No-Fly Zones, Out of Confined Area & Loops

The method followed to avoid no-fly zones has been done by checking in each movement that the drone does. If there is an intersection with a no-fly zone perimeter line, the angle of the movement will immediately be corrected. The angle will be generated using the current position of the drone and the final coordinate to be reached. In this way, the final location will be reference and the angle α will try pointing always to move into that direction. Knowing this, it has been decided that going backwards would be counter efficient and will create a loop as the following movement would be returning to the conflict position. Therefore, the approach chosen to correct angles is to check, the angles in the range +/- 120º from the suggested initial one α. The process will check first all the angles to the left till $\alpha + 120º$, increasing the angle 10 by 10; and if no valid option is found, the angles to the right will be checked, $\alpha - 120º$, reducing the angle -10 by -10.

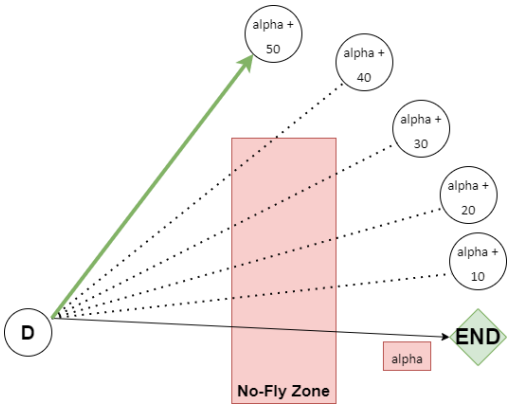

Figure 1. Diagram of the drone avoiding a no-fly zone

Finally, it should be added that if the drone does not find any direction to go to, the system will operate the initial α move and searches an option to get out. As all these steps are simulated, in the worst-case scenario, the drone will go into the no-fly zone and try getting out. This will cause the drone to intersect continuously with the inner lines of the no-fly zone, generate a loop.

To avoid loops, the simulation stops immediately when the method *go to()* has operated the maximum number of available steps plus 1 (15001). This breaks possible loops generated if the drone gets stuck in an area with no way out. The path is later one compared with the number of available moves (maximum 1500) and immediately the order will be skipped. This method avoids those orders that could provoke runtime exceptions.

It should be remarked that when a movement is going to be done and the angles are being checked, the new position is also checked to ensure that is confined. Therefore, all the implementation to avoid the no-fly zones and the logic used to break loops is applied directly over this restriction and when this issue arises, the drone directly does not effectuate the movement.

Returning to the Base

As mentioned in the entry **Main Aim, Movement Restrictions, & Confined Area** the drone must ensure it can return to Appleton without running out of battery (before it reaches 1500 movements effectuated). The drone will simulate the path to the base, similarly to a delivery, but not going through restaurants. The same methods *get path to landmarks()*, *get landmarks()* and *go to()* will be used in this process. Finally, the path simulated is committed into the flightpath database table and into the GeoJSON file.

Writing a GeoJSON File

When the drone returns to the base, the system must generate a GeoJSON file which will contain a FeatureCollection with a LineString with all the coordinates through which the drone has flown. This file will be named "drone-DD-MM-YYYY.geojson", where DD is the date, MM is the month, and YYYY the year input by the user. The program will use the method *write GeoJSON()* of the **GeoJSON Handler** class.

Drone Flight Figures

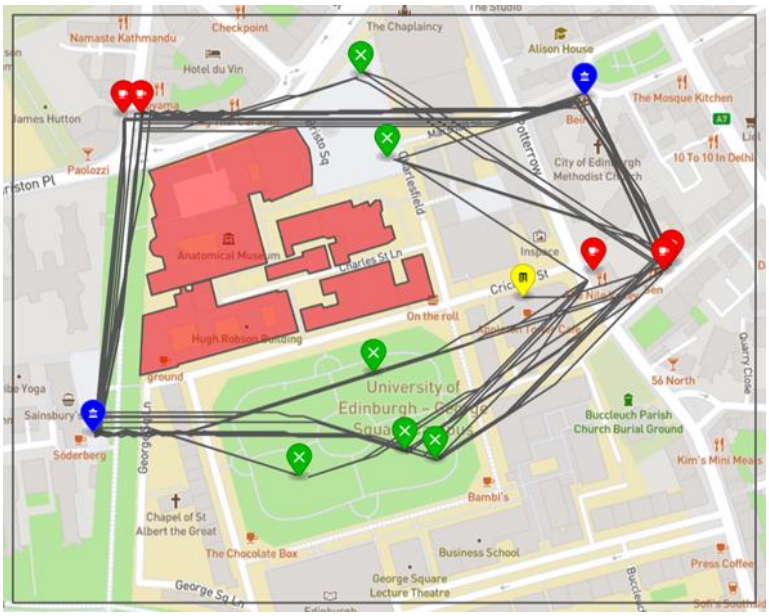

Figure 2. Deliveries completed on the 11th/06/2023.

- Deliveries 21/21
- Movements 1446
- Money made: 19785

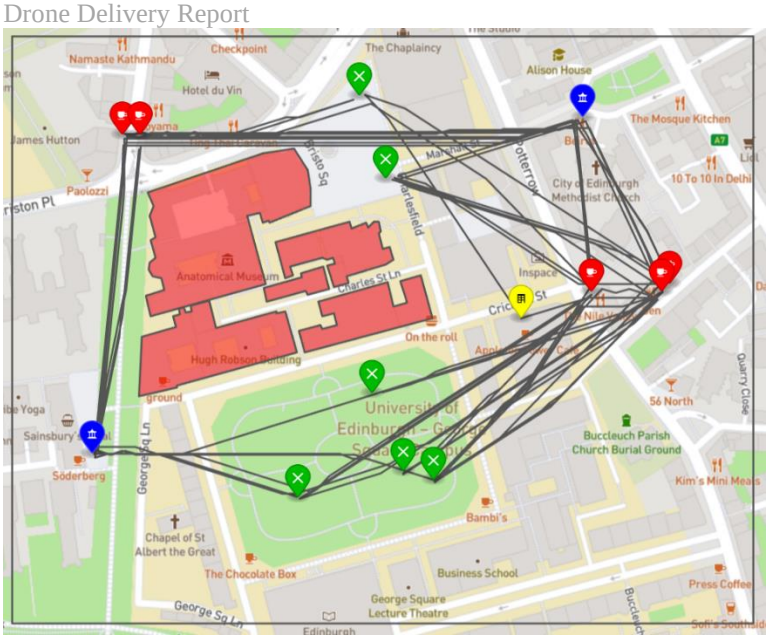

Figure 3. Deliveries completed of the 11th/12/23

- Deliveries 25/27
- Movements 1500
- Money made: 19950

Generating Submission Files

To improve the methodology to generate the files from which the results and statistics will be done & easily generate the 12 submission files, a python script **files and statistics generator.py** has been written to run the jar file of the project. This script will contain multiple methods that allow running the program in a loop and generate files where DD = MM of a year, files of all months for a given DD, files of all a month. The ending results of each date (completed deliveries out of total, total number of steps, money made out of total possible, and percentage of money achieved) is also stored into a "drone-DD-MM-YYYY-results.txt". Important notice, the ports used in the file are the default ones. Consider changing them if necessary. The main function is set to generate the submission files requested for the years 2022 and 2023.

Results and Statistics

To get an accurate idea of the performance of the algorithm, the student took three sampled average percentage monetary values delivered by the service:

Getting the sampled average percentage monetary values of all the 11ths of the database. (all months & all years):

Total number of dates: 24 (all 11-MM-YYYY)
Total number of deliveries completed: 358 / 372 – 96%
Total value of the deliveries: 299641 / 303736 (pence)
sampled average percentage: 99%

Getting the sampled average percentage monetary values of all the dates in December 2023 (the month with more deliveries):

Total number of dates: 31 (all DD-12-2023)
Total number of deliveries completed: 713/ 837 – 85%
Total value of the deliveries: 617019 / 660381 (pence)
sampled average percentage: 93%

Getting the sampled average percentage monetary values of all the dates from the last 2nd half of 2023 (1st June – 31st December):

Total number of dates: 214 (all DD-MM-2023, MM from 6 to 12)
Total number of deliveries completed: 4705/ 5137– 92%
Total value of the deliveries: 3956095 / 4100616 (pence)
sampled average percentage: 97%
Movements Done: 309355
Interesting fact: the runtime of the process was of exactly 8 mins (start 16:25:00 – end 16:33:00):
June 67s, July 67s, Aug 70s, Sep 67s, Oct 72s, Nov 68s, Dec 69s (with slight possible fluctuation between them).

In any of these processes, the drone algorithm got stuck in a loop generating runtime exceptions.

References

- Informatics Large Practical, Stephen Gilmore and Paul Jackson – School of Informatics, University of Edinburgh, 2021 – 2022.
- GeeksforGeeks. (2013). *How to check if two given line segments intersect?* [online] Available at: https://www.geeksforgeeks.org/check-if-two-given-line-segments-intersect/ [Accessed 22 Nov. 2021].
- docs.mapbox.com. (n.d.). *services-geojson API.* [online] Available at: https://docs.mapbox.com/android/java/api/libjava-geojson/5.8.0/ [Accessed 19 Dec. 2021].