Informatics Large Practical Coursework 2

s1963915

December 2021

Section 1: Software architecture description.

The software architecture of the application consists on a collection of 13 different classes, those being App, Database, Drone, GeoJSON, HTTPClient, LongLat, MenuItem, Menus, NoFlyZones, OrderDetails, OutputFiles, Restaurant and What3Words. These classes form a Java application that takes as inputs 5 different values: the day, month, and the year, the web server port number and the database port number, and outputs a .geojson file detailing the deliveries of the drone for that given day and two databases logging the relevant information about the drone's path.

App class

This is the main class of the java application. It receives as inputs 5 different values. The first three input values correspond to the day, month and year representing the date for which we want to simulate the deliveries, and the last two correspond to the web server port number and the database port number. Once the inputs are received, the class then checks that the user inputs are valid. Then, if they are valid, the App connects to the database, and using the *builDate* method to constuct the relevant date, reads from the database obtaining a java ArrayList with all of the orders that have been ordered for that date. It is important to note that we obtain the orders as OrderDetails objects, where for each order we read from the database tables the order number, the customer, where we deliver the order to, and the items forming that order, and then we call the *setOrderDetailsFields* method to calculate and complete the remaining parameters of the OrderDetails folder. Once we have the complete information about all the orders in the database for the given date, we then create a NoFlyZones object, and we then read and store from the web server all of the no-fly zones in the parameter noFlyZonesPoints. It is important to note that we need to load and store the information about the orders and the no-fly zones, as we then proceed to create a new object of type Drone, and it has all of the no-fly zones and orders of that day as parameters. The Drone object will calculate the necessary resulting information about the orders that the drone is going to do that day, the route that it is going to take, etc. Using that information we can then write the geojson output file, using the method *writeGeoJSONFile* as well as populating the output database tables *deliveries* and *flightpath* with the methods from the Database class *writeDatabaseTableDeliveries* and *writeDatabaseTableFlightpath.*

Database class

This is the class responsible for handling any queries regarding the database. When calling the constructor, it initialises a *java.sql.Connection* with the database, that is later used for all of the methods of the Database Class. This way, we don't have to waste resources connecting to the same database several times. The first method of the Database Class is the *getOrderDetails* method, that, given a date, it access the database and reads from the orders table. Thus, for each order in the orders table, we extract the fields for the *orderNo*, *customer*, and the location to where we deliver *deliverTo*, and we store every order as a *OrderDetails* object populating the aforementioned fields. A similar purpose serves the second method in the *Database* class, *getCompletedOrderDetailsList*, which iterates through the orderDetails table, and by iterating through the table, for every order number that we have, it completes the items parameter in each *orderDetails* object. Finally, we have two other methods in the *Database* class: *writeDatabaseTableDeliveries* and *writeDatabaseTableFlightpath*. Unlike the first two methods, these are responsible for writting the two tables, *deliveries* and *flightpath*, that we have to output. They do this by iterating through some of the results produced by the *Drone* class, representing the information about the coordinates and angles that the drone route consists of, and the information about the orders delivered by the drone.

OrderDetails class

The *OrderDetails* class helps us represent the orders. Every order that we have is represented by an *OrderDetails* object. The OrderDetails class has several parameters with self-explanatory names, those being *orderNo*, *customer*, *deliverTo*, *items*, *price*, *deliverFrom*, *deliverToLongLat*, and *deliverFromLongLat*. It is important to note that *deliverTo* and *deliverFrom* are in What3Words format, while *deliverToLongLat* and *deliverFromLongLat* are in LongLat form. It is also worth noting that *deliverTo* and *deliverToLongLat* consist of just one element each, while *deliverFrom* and *deliverFromLongLat* can have more than one element, as we can pick up orders from more than one restaurant.

Then, the constructor of the *OrderDetails* class is built with the parameters *orderNo*, *customer*, *deliverTo*, and *items*, which, as explained before, are obtained from the *Database* class, and then we use the *setOrderDetailsFields* method to complete the rest of the parameters. This method calls the *setPrices* method to get the *price* parameter using *Menu.getDeliveryCost*, the *setWhat3WordsLocations* that completes the *deliverFrom* parameter by adding the restaurants that have the ordered items (using the *Menu.getRestaurantLocationOfItem* method); and the *setLongLatLocations* method, that completes the *deliverToLongLat* and *deliverFromLongLat* parameters by translating the What3Words into LongLat objects by using the *HTTPClient* method *translateLocation*.

LongLat class

The *LongLat* class is how we represent the coordinates of a point. This class has two fields, *longitude* and *latitude*, representing respectively the coordinates of the longitude and latitude of the *LongLat* point. We construct the *LongLat* class by providing the two aforementioned parameters to the *LongLat* constructor. The *LongLat* class then has an important method called *isValidMovement*, that, given the next position and the noFlyZones, it tells us whether the drone can make that move from the current position. This method uses the *isConfined* method, that tells us if that point is within the confinement zone, and the *intersectsWith* method, that we use to check if the next movement crosses a line marking the boundary of a no-fly zone. The *LongLat* class also has other useful methods that we use to move the drone, such as *distanceTo*, that gives us the pythagorean distance to a given *LongLat* point; *closeTo*, that tells us if the point is close enough to the destination to consider that it has arrived to that destination; and the *nextAngle* and *nextPosition* methods, which calculate the angle needed and the *LongLat* location where the drone will be after performing a move with the calculated angle. Finally, we also have a helper method *translateLandmarksToLongLat* that transforms lists of *mapbox.geojson.Point* into lists of *LongLat* so that we can work with all of the methods we have implemented in this class.

What3Words class

The What3Words class indicates the other way to represent a location. It has no methods and only one attribute, *coordinates*, which is necessary to use to parse the data for the location of the restaurants. This is done in the *translateLocation* method in the *HTTPClient* class, where we use this attribute to translate a *What3Words* location into a *LongLat* location.

HTTPClient class

The *HTTPClient* class is responsible for handling all queries and connections to the web server. It has an static final attribute *java.net.http.HttpClient*, which is responsible for sending the HttpRequests and retrieving their HttpResponses for all of the main methods of this class. Among the methods of this class, there is the previously mentioned *translateLocation*, where we connect to the *words* folder in the web server to translate the *What3Words* locations into *LongLat* locations. We also have the methods *getNoFlyZones* and *getLandmarks*, that connect to the *buildings* folder in the web server to obtain the noFlyZones and the landmarks. Finally, we have the method *getRestaurantRequest*, which access the remaining folder *menus* in the web server, and retrieves the list of type *Restaurant* containing the name, location and the menu of every restaurant, as explained in the next class.

Restaurant class

We use this class to read from the web server *menus* folder. The *menus.json* document that is inside the *menus* folder has fields *name*, *location*, and *menu*, and thus to deserialize the menus JSON record to the Java *Restaurant* class, we have the attributes *name*, *location*, and *menu* in the Restaurant class. Thus, the *Restaurant* class is used to store the information from the *menus.json* file, and represent a restaurant with name *name*, location *location*, and menu *menu*. Finally, it also has a method *getMenuHashmap*, that we use to store the information about the menus in a HashMap, to iterate through the menu of a restaurant more efficiently.

MenuItem class

While the *menus.json* document that is inside the *menus* folder in the web server has attributes *name*, *location*, and *menu*, the attribute *menu* is a list with attributes *item* and *pence*. Thus, to be able to deserialize from the *menus.json* document the contents of the *menu* attribute, the class MenuItem is created, whose attributes are exactly *item* and *pence*. Thus, the *MenuItem* class stores the information about an item, with item name *item* and its price in pence *pence* of an item in the *menu* attribute of the *Restaurant* class.

Menu class

The *Menu* class contains useful methods to deal with objects of type *Restaurant*. We call the constructor of the *Menu* class by providing the machine name and the web server port, and the constructor calls the method *getRestaurants* to obtain the list of *Restaurant* containing all of the available restaurants. This is obtained by calling the previously mentioned *getRestaurantRequest* method from the *HTTPClient* class. We then use this list of all objects of type *Restaurant* in the methods *getDeliveryCost* and *getRestaurantLocationOfItem*, which are methods that we use in the *OrderDetails* methods *setPrices* and *deliverFrom* respectively.

NoFlyZones class

This class is used to represent the no-fly zones that the drone can not hover through. This class is called from the *App* class, and its constructor takes as parameter the *HTTPClient.getNoFlyZones* method discussed before, which gives us the no-fly zones as an ArrayList of *mapbox.geojson.Polygon* objects, and the constructor then calls the *setNoFlyZonesPoints* method to set the *noFlyZonesPoints* attribute. The *noFlyZonesPoints* attribute stores the no-fly zones as an ArrayList of ArrayList of LongLat, where each sub-ArrayList represents a collection of points, each pair representing the endpoints of the one of the lines representing the borders of a particular enclosed no-fly zone area, and the main ArrayList thus represents the endpoints of all the different straight lines representing the borders of all no-fly zones.

Drone class

This class is used to represent the behaviour of a drone. The constructor of the class is called from the *App* class, and we provide as arguments *orderDetailsArrayList*, which is the list of *OrderDetails* representing all of the orders that have been placed for that day, and the no-fly zones *noFlyZonesPoints*. The *Drone* class then has three main methods that handle the data, each of which is called in the constructor. The first one is *setOrderDetailsToDo*, which decides from all of the possible orders which orders are going to be done (which could be all orders, if it has enough moves). It thus populates the attribute *orderDetailsToDo*, which contains the orders that are to be done. The second method is *setCoordinatesToVisit*, which takes the list of the orders that the drone has decided to do and populates the *coordinatesToVisit* attribute, which represents the list of coordinates (for the restaurant(s) and for where we need to deliver the order) the drone has to visit to complete those orders. Finally, the third method is *setRoute*, which given the list of coordinates to visit, it constructs the relevant route to visit all the given coordinates to visit, taking into account the noFlyZones and not doing illegal movements, and populates the *route* attribute. Thus, the *route* attribute contains a list of LongLat objects that are each of the coordinates the drone is visiting, including the initial coordinate at Appleton Tower's location, and the final coordinate close to Appleton Tower. The *Drone* class also has multiple other helper methods. Some of the more relevant ones are *canReturnToAppleton* and *canPerformNextOrder*, which always check respectively (before commiting to perform an order) if we were to do that order, if we would have enough moves to return to Appleton and whether it would be physically possible for the Drone to do so (if the drone ends up in a point where there are no useful Landmarks nearby, and thus would not be able to return to Appleton without crossing no-fly zones). Other relevant methods would include *travelToDestination*, which returns the route for travelling from one given point to another, or *getMovesAngles*, which helps us populate the *angles* attribute, an ArrayList of Integers of the length of the number of moves we do, and where for the i^{th} move the drone does, the i^{th} element of *angles* indicates the angle that the drone has travelled with for that movement. Finally, it is also worth mentioning the *orderNumbers* attribute, which also has the same length as the number of moves, and for the i^{th} move the drone does, the i^{th} element of *orderNumbers* indicates the order number that move corresponds to.

GeoJSON class

The *GeoJSON* class only has one method, *translateRouteToGeoJSON*, which takes as parameter the *Drone* object, and it obtains the *route* attribute explained before, which consists of an ArrayList of LongLats representing the locations of each coordinate the drone visits, and transforms it into a *mapbox.geojson.featureCollection* of *Linestring* objects, that is, it connects all of the coordinates of the points transforming them into a collection of lines that we later use in the geojson output file. Even though this class is currently short, it is still justified to include this class as this is the best way to handle *mapbox.geojson* objects, and if this project is expanded when the drone enters into service, then all *mapbox.geojson* objects could be handled here.

OutputFiles class

Finally, the last class to mention is the *OutputFiles* class, which is responsible for handling the files that out java application outputs. the only method in this class is the *writeGeoJSONFile*, which as mentioned before is called from the *App* class, and given a day, month, year and a drone, it produces a *.geojson* file representing the path of the drone. We use the previously described method *GeoJSON.translateRouteToGeoJSON* to generate the lines representing the route, and the *writeGeoJSONFile* produces the filename and writes the *.geojson* file into the current working directory.

Section 2: Drone control algorithm

Algorithm

The algorithm that is being used has a simple objective: maximise the metric pricePerDistance, which we obtain by dividing the total price we receive by performing a given order, by the distance we need to travel to complete the order. Thus, the general idea of the algorithm would follow the following structure:

1. Start with the full list of orders to do for a given day that have been read from the database.
2. Set the number of moves that have been used to 0.
3. Set the starting location of the drone as the Appleton Tower.
4. For every order remaining, calculate the *pricePerDistance* metric, by dividing the price that we would obtain by performing that order and dividing it by the distance that the drone would travel by starting in the original location, travelling to the restaurants and then delivering the order to the drop-off location.
5. Pick the order with the highest *pricePerDistance* metric, and perform that order the next.
6. Add the number of moves performed by doing the order to the number of moves that have been used.
7. Delete the order performed from the list of orders to do.
8. Set the starting location of the drone as the location where the drone has just dropped off the order.
9. Repeat steps 4-8 until:
   (a) The list of orders to do is empty.
   (b) The amount of moves necessary to perform the next order with the highest *pricePerDistance* metric and then return to the Appleton Tower is larger than ((Maximum number of moves) - (number of moves that have been used)).
10. Return to the Appleton Tower.

Comparison of algorithms

This algorithm has the advantage that it focus first on the most profitable routes, and then, if there are enough moves left, it continues doing the less profitable routes. This algorithm, can be considered a greedy algorithm as it always picks as the next order the order with the highest *pricePerDistance*. However, it has a great advantage compared to other traditional greedy algorithms. For example, compared with a greedy algorithm that would first calculate the *pricePerDistance* for every order and then perform the orders in that order (without checking where the drone would be when the prior order ends and taking that in its calculations), may cause the drone to waste moves by, for example, picking orders that are on the opposite side of the map, as we would not take into account the distance spent travelling to the next order. Another greedy algorithm, which would just do first the orders that pay the most, would also perform worse than the our algorithm as it would also not take into account where the drone ends up after each order. In the extreme case, this last algorithm would also prefer to make one order of 1500 moves that pays 1500, than to perform 1500 orders of one move each, each paying 1000. Thus, the algorithm that has been implemented, while not being very computationally expensive, gives us a good approach to make the most profitable routes while encouraging the drone to pick up orders that are closer to where the drone finished the prior order, and therefore wasting less moves. This algorithm therefore would return a higher sampled average percentage, while being computationally relatively inexpensive, which is important as it has been emphasised that hungry students should not be waiting long for their orders to be delivered.

Satisfying the constraints

The first stipulation for the algorithm is that the drone can make at most 1500 moves before running out of battery. This condition will always be fulfilled by the implemented algorithm, as we specifically check before performing any order if we are going to have enough moves to return to Appleton after performing that order. If the drone is not going to be able to return, then that order will not be performed, and the drone will choose to return to Appleton Tower for a recharge. Therefore, the condition that the drone should return close to that location is also satisfied.

Another constraint is that every move when flying has to be a straight line of length 0.00015 degrees, and with an angle multiple of 10 between 0 and 350. These conditions are fulfilled as the drone will moves with the *LongLat.nextPosition* method, which moves the drone in a straight line of length 0.00015 degrees, and the angle is given by *LongLat.nextAngle*, which strictly returns an angle between 0 and 350 and multiple of 10.

Third, the drone must hover for one move when collecting the orders and when delivering the lunch, with an angle of -999. This is achieved by manually making the drone stay in the same place for one move when arriving to a destination, and returning the angle value of -999.

Avoiding no-fly zones and staying inside the confinement area

The algorithm always stays inside the confinement area and avoids the no-fly zones. This is due to the fact that, before committing to any move, we first check if our next move is inside the confinement are and if we are not going to cross a No-Fly zone. Thus, if a move is illegal it will not make the move, although it is important to point out that the route has been planned before-hand making sure that the drone can indeed make the given route, so the drone will never waste moves moving to a direction to find out mid-way that it can not continue.

Second, it is important to note how the no-fly zones are checked to make sure that we are not crossing a No-fly zone. A common way of checking if the drone goes into a no-fly zone would be to check if the next move we want to make is inside the no-fly zone or not. However, this method would not take into account that the drone might briefly fly over a no-fly zone, even though when the move finishes the drone is outside the no-fly zone. However, the drone must not under any amount of time fly over the no-fly zone. Thus the method implemented to check if the drone enters the no-fly zone is the following. We first store the no-fly zones not as geographical areas, such as polygons, but like arrays of points, where each pair of consecutive points form the boundaries of the lines forming the no-fly zones. Then, every time we want to perform a move, we iterate through every pair of points representing the lines of the boundaries of the no-fly zones, and we use the *LongLat.intersectsWith* method to check, for each boundary line, if the hypothetical line connecting the prior and next position of the drone would intersect with the line representing the boundary of the no-fly zone. This is indeed more computationally expensive, but as the no-fly zones do not contain many boundaries and the no-fly zones are not expected to increase exponentially, it is a good way to check if the next move intersects with a no-fly zone.

Taking into account the variability of the initial conditions.

It is specified in the coursework that the locations of the no-fly zones and the landmarks are the result of our current best guess, and that they may change in the future. Thus, a great emphasis has been placed when designing the application to make sure that the drone will work as intended with any no-fly zones and with any amount of landmarks. Thus, if the landmarks or the no-fly zones would need to change, they could just be changed in the web server and the java application would remain working as intended, given the new parameters.

The code has been optimised to take care of the circumstance when we have a different number of landmarks than 2. Thus, the way the code works, is that, when we are not able to travel directly from one point to another, we try to use a landmark and for every available landmark, we would try to go to that landmark and then to the intended destination. If we would be able to go with at least one of the landmarks, we would then pick the shortest path and proceed as usual. However, the location of the landmarks do not guarantee that we will be able to actually arrive to our intended destination: it may be that even when using the landmarks we would not be able to arrive to our destination. In that case, the program would know that we would not be able to reach that point, and thus we would not be able to go there. The program would plan for this eventuality before-hand, as before committing to do an order, the program also checks with the *canPerformNextOrder* method whether we would be able to perform the order, and whether, if after performing that order, we would then be able to return back to Appleton. This has also the purpose to avoid the case where we can reach the place where we have to deliver the order, but then due to the location of the landmarks, it would not be able to find a path back to Appleton.

Therefore, the application would work when either more landmarks are added, or even in the case when just one landmark is left (it is specified that the landmarks are not null, there will always be at least one landmark), in which case less orders will be able to be completed but the algorithm still works as intended. Special care has even been taken care of the special case where the only landmark is the Appleton tower: in this case, for example, the 2022-02-04, the drone would be unable to fulfill any of the 5 orders placed that day, as it can not reach the necessary coordinates to deliver the orders without any additional landmarks. Thus, the program would also be able to handle the unlikely issue that no orders can be fulfilled.

Finally, it is worth mentioning that due to the way the program handles the possibility of orders not being able to be fulfilled, it would also not be affected by a different set of no-fly zones. The drone would still not access any place without being sure it can go there first and then have a way to come back. And even if the no-fly zones would prevent the drone from making any move at all, the program would still be able to handle that situation. However, it is important to note that the program does rely on the assumption that at most two shops will be visited, however, according to the coursework this assumption will hold so the application should work as normal.

Two examples of the algorithm.

We can see in the two figures included below two examples of paths generated by the drone on two different days. Figure 1 represents the flight path corresponding to the date 05-08-2023 and Figure 2 represents the flight path corresponding to the date 05-08-2023.

The Figure 1 is an example of when the drone could not complete all of the given orders. That day there were 23 possible orders that the drone could have done, but the drone only managed to complete 21 of those orders before returning to Appleton Tower. It made 1473 moves, and it managed to obtain an average percentage monetary value metric of 96.8%. This is much more than the proportion of orders the drone made (91%), thus suggesting that the drone is effective in performing the most lucrative orders first. From the figure, we can observe that the drone ends up close to each location, and we can also observe that the drone returns close to Appleton in the end. Finally, it is also worth noting that the two landmarks are being used; and that the path of the drone does not even get close to the NoFlyZone, and always stays confined in the confinement area.

Figure 2 is however an example of the much more common occurrence of when the drone is able to complete all of the orders before returning to Appleton. This time, the drone completes 20 out of the 20 possible orders, and returns to Appleton Tower after performing 1457 movements. As it has completed all of the moves, we obtain an average percentage monetary value metric of 100%. Once again, we can check in that all of the paths drawn make sense, if we zoom in we can see that again it returns close to Appleton Tower in the end and that it does not cross the NoFlyZone at any moment, as well as the drone always staying within the confinement area.

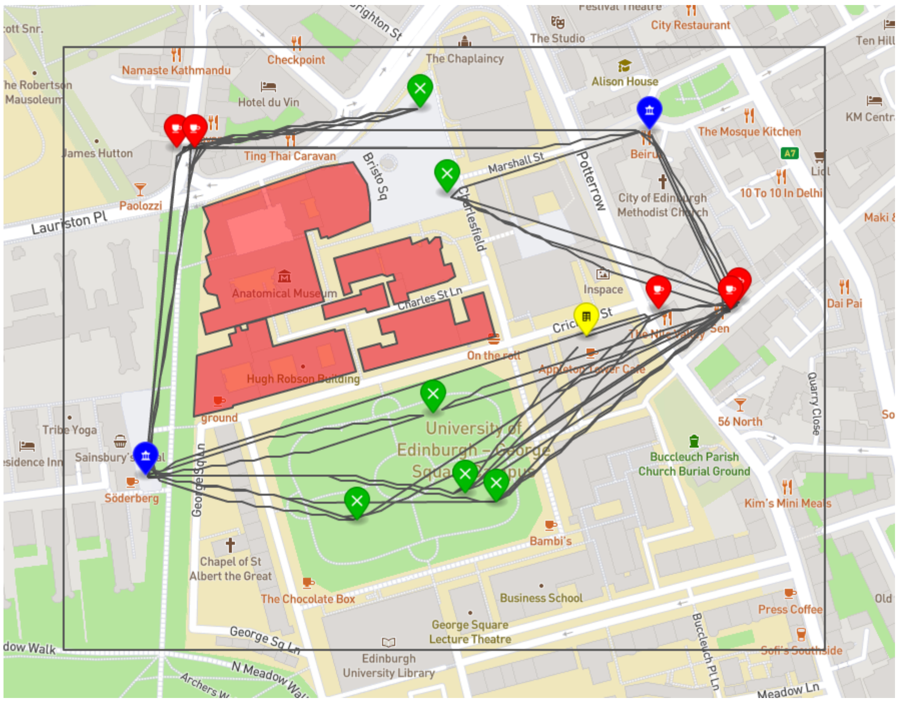
Figure 1: Flight path corresponding to the 05-08-2023

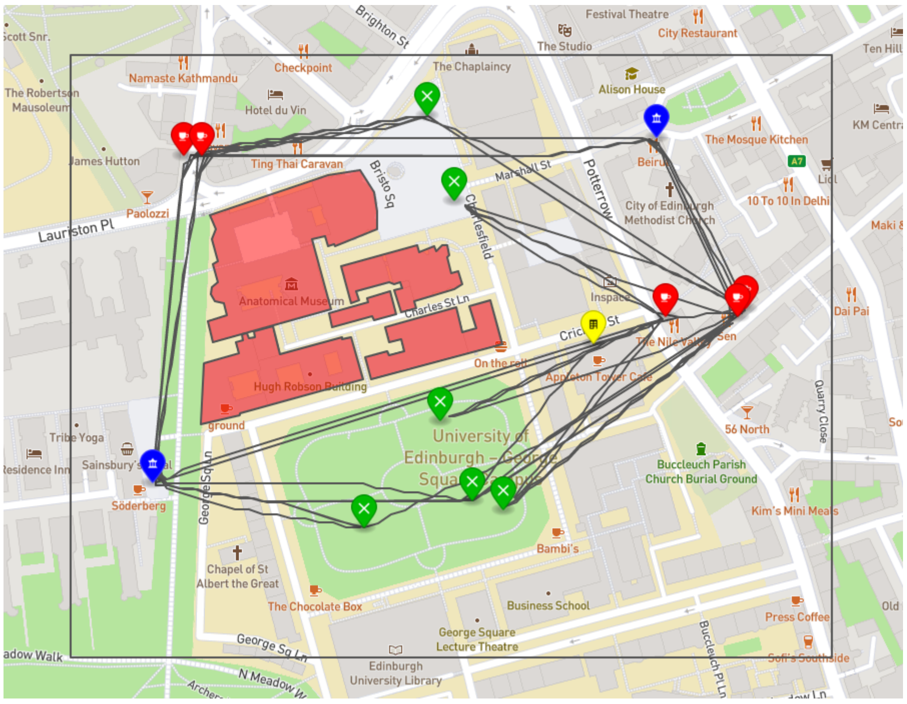
Figure 2: Flight path corresponding to the 07-05-2023