s1960565 – ILP Report

Software Architecture Description

My application is made up of 10 classes:

- App
- Buildings
- Drone
- ItemDetails
- LongLat
- Menus
- Orders
- ShopDetails
- W3wDetails
- Words

Class descriptions:

App –

This class is required because it is home to the main method. This is where we take in the required program arguments to run our application. These include the date, month, and year that we are providing our service for, the port on which the web server is run on, and the port on which the database server is run on. In the main method in the app class is where we create instances of most classes and actually put them to use.

Buildings –

This class is required to be able to work with information about the buildings that we are expected to consider. These buildings include no-fly-zones and landmarks. Shops are considered in the Menus class. The information about these buildings is retrieved from the web server, and we keep our relevant information as fields called nfzEdges and landmarkPoints. Because our application is run only to complete a day's worth of service, it is expected that information about these buildings

will not change during execution, thus it is enough to only create a single Buildings object throughout our program that we can continuously use to access information about the buildings we are interested in.

Drone –

This class is required to be able to represent and create a real drone object that we can work with. The attributes of a drone object include the position of the drone, the amount of battery the drone has left, and the number of moves the drone has made. We also have fields that are lists of positions and angles that is important to write to the flightpath table.

Menus –

This class is required to be able to represent every shop and their menus, which is information that we need to provide the service. All of this information is on the web server as JSON files that we need to parse and retrieve. This information includes details about the shops such as their name, location, and menu as well as details about the items on the menu such as their name and price. The key relevant information is stored in fields itemToPrice, itemToShop, and shopToWords which are all useful hash maps. Since it is expected that information on shops and menus will not change during program execution, it is enough to create a single Menus object that gives all the information for us, that we can use throughout the runtime.

ShopDetails –

This class is required in order to parse the JSON records described in the Menus class as java objects. This JSON file consists of a number of shop details, so we created this class to represent exactly that. In the JSON file, the details include the name of the shop, the location of the shop and the menu of the shop. So, we set these as attributes of the class. The name and location of the shops are both string objects however the menu is of type ItemDetails.

ItemDetails –

This class is required to be able to represent the menu attribute of a ShopDetails object. The menu attribute is just a list of ItemDetails objects. As the name suggests, an ItemDetails object represents the details of an item. The attributes are item name and item price because that is how they appear in the JSON file.

LongLat –

This class is required to represent positions on the map. It has a number of uses, for example the drone's position attribute is of type LongLat, most of the shops and landmarks positions eventually gets parsed as a LongLat object so that it is easy to work with. A LongLat object has two attributes that is needed to specify a location, which is longitude and latitude.

Orders –

This class is required to be able to represent all the orders that were placed on the day we are interested in. All of this information is stored on the database server. So, the attributes of an Orders object require the port on which the database server is running on and the date of which we want to retrieve the orders from. Because we don't need to consider if more orders will be placed during run time, we only need to create a single Orders object which will hold information about all the orders placed on the date we are interested in. This information includes the order numbers, the delivery locations and the items that were ordered, of orders.

Words –

This class is required to represent the location of a What 3 Words address. Because the contents of this information are located in a different place on the web server, I thought it would make more sense to create a new class for it. Each JSON file on the web server contains information on only one W3W address, therefore we will have to create more than one Words object if we want to retrieve the location of more than one W3W address. A Words object requires the port the web server is running on (because all of the information is on the web server) as well as a string array containing three words, which combines to make a W3W address. The reason why it is split up into an array is because of how the file paths are on the web server. The purpose of the Words class is to get the location of a W3W address, so we create a field called coordinates which stores this.

W3wDetails –

This class is required in order to parse the JSON records described in the Words class. Each JSON file has a record that contains lots of irrelevant information to us, however there is one piece of information we need, which is the coordinates record. Thus, a W3wDetails object has only one field , which is coordinates.

Drone Control Algorithm

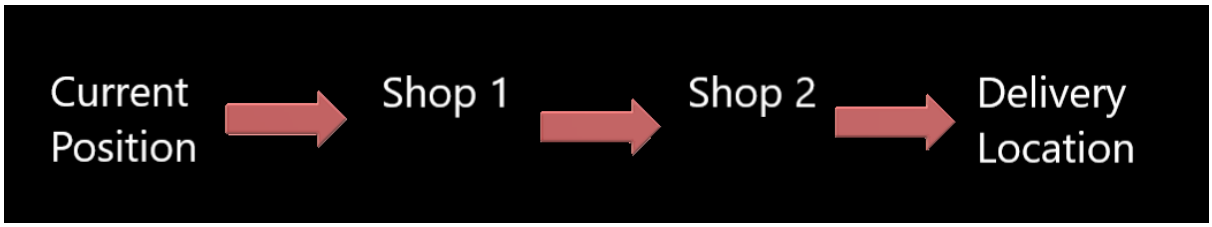
The way my main drone is controlled is through 2 phases: The pre algorithm, and the algorithm.

Pre Algorithm:

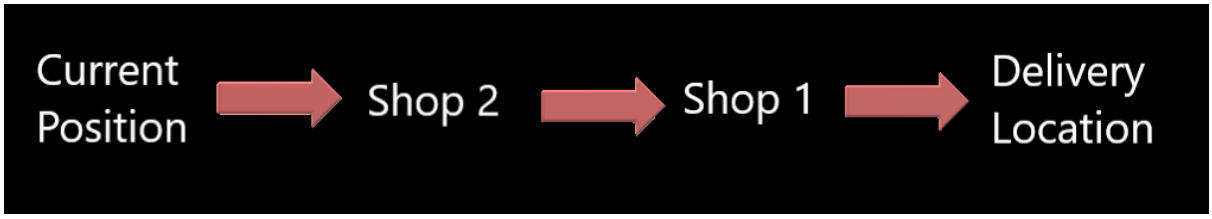
- First of all, we sort the orders so that the value of the orders is descending. We prioritise value of orders here to help maximise the drone's score on the sampled average percentage monetary value metric.
- We then iterate through the list of sorted orders. For every order we:

    - Precompute the best route to take to complete this order using a brute force approach, the best route being the route which minimises the number of moves made. Since there is a maximum of only two possible routes (as shown below) the brute force approach will not be too costly here. There is also the case where the order only contains items from a single shop, in which case we only have one route to take.

Route 1:



Route 2:



Route 3 – Trivial Solution:

- After we have the most cost effective route, we check (using a dummy drone) whether our drone has enough battery to take the route as well as return back to Appleton Tower from the delivery location. We check this because if it does not have enough battery to do so, we will not be able to return back to Appleton Tower after we call the main algorithm for the order in question.

    - If our drone does have enough moves to do this, we call the algorithm for this order.
    - If our drone does not have enough moves to do this, we simply ignore this order. This allows for the possibility of orders of less value and those that require fewer moves, to be completed, although not guaranteed.

  After we have iterated through each order, we will have either completed the deliveries of every order or reached a point where we don't have enough battery to deliver any remaining order.

- We then call a special version of the algorithm, which returns the drone back to Appleton Tower.

Here is an example of the above behaviour (values are not to scale of real values):

Orders sorted by value (1 is most expensive and 3 is least)	Battery of drone	Number of moves required for best route + return to Appleton Tower	Should we start completing the order?
1	1500	900 + 100	Yes
2	600	600 + 150	No
3	600	500 + 100	Yes

Order 1 –
We can complete order 1 since the battery is greater than (or equal to) the number of moves required to take the best route and return back to Appleton Tower (1500 >= 1000), this will leave us with 600 battery left, since we do not return back to Appleton Tower (which would take us 100 moves) yet because there are more orders left to check.

Order 2 –
We can not complete order 2 since the battery is less than the number of moves required to take the best route and return back to Appleton Tower (600 < 750).

Order 3 –
We can complete order 3 since the battery is greater or equal to than the number of moves required to take the best route and return back to Appleton Tower (600 >= 600). Since this is the last order in the list, we can return back to Appleton Tower after completion.

Algorithm:

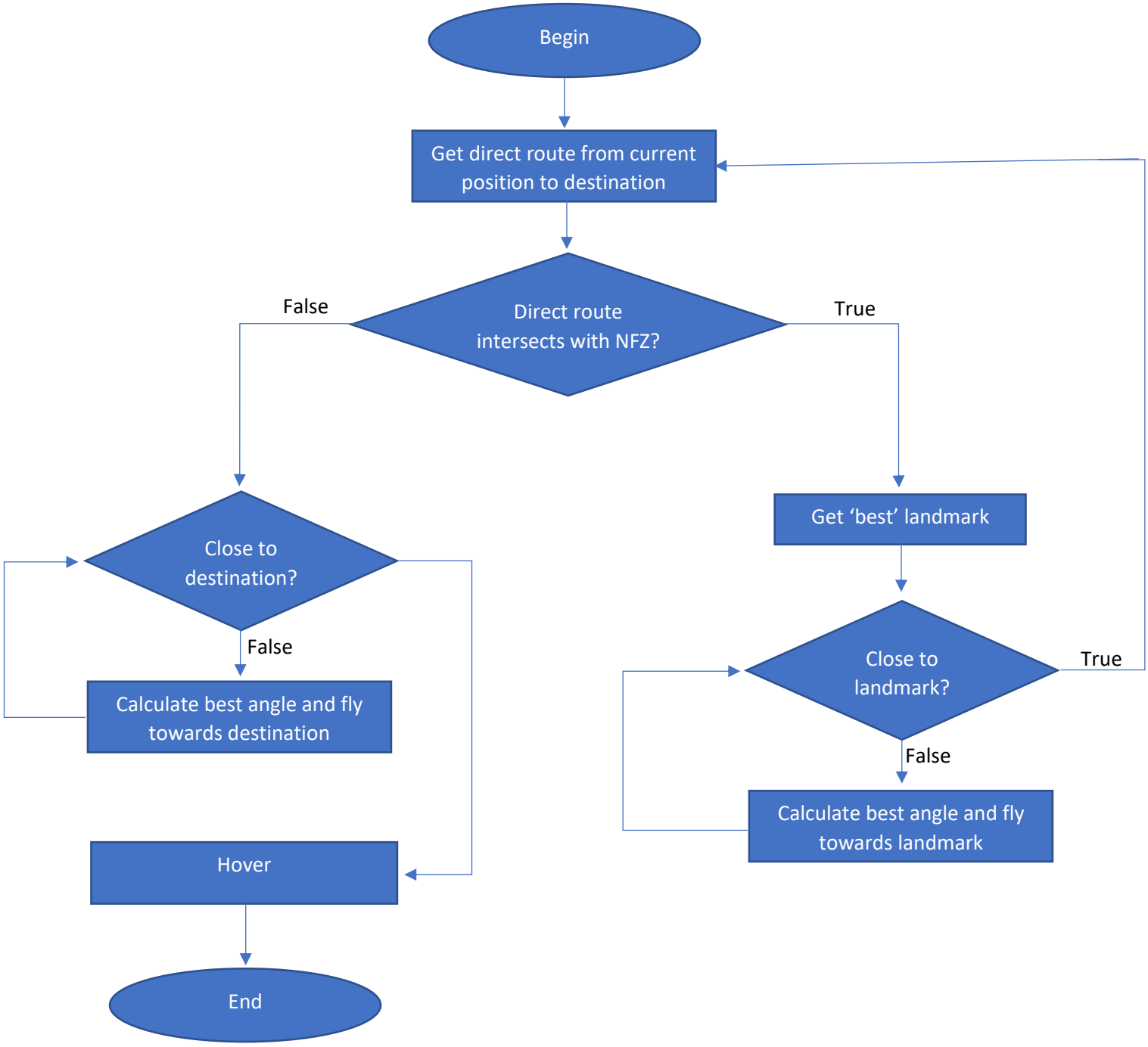
Note, this algorithm is called once for each order to make the drone actually move to where it is supposed to be going.

- We iterate through every destination (shops and delivery location) in the best route calculated in the pre-algorithm phase. For each of these destinations, we:
    - Draw a straight line (of any angle, which doesn't need to be a multiple of 10) from our current position to the destination. We then check if this straight line (direct route) intersects with any part of the no-fly-zones (line intersection). We repeat the following until this direct route does not intersect with a no-fly-zone. So, if it does intersect with a no-fly-zone, we:
        - We must get the closest landmark to the destination such that there exists a direct route to from our current location. This landmark will be our temporary destination that we need to divert towards. We repeat the following until our drone is close to the landmark:
            - Calculate the best angle to take to travel toward the landmark and fly with this angle.
    - We now have a direct route to the destination, so we can simply repeat the following until we are close to the destination:
        - Calculate the best angle to take to travel toward the destination and fly with this angle.
    - We have arrived at the destination (which is either a shop or a delivery location), so we must hover for one move.
- Once we have iterated through every destination, we will have completed the order.

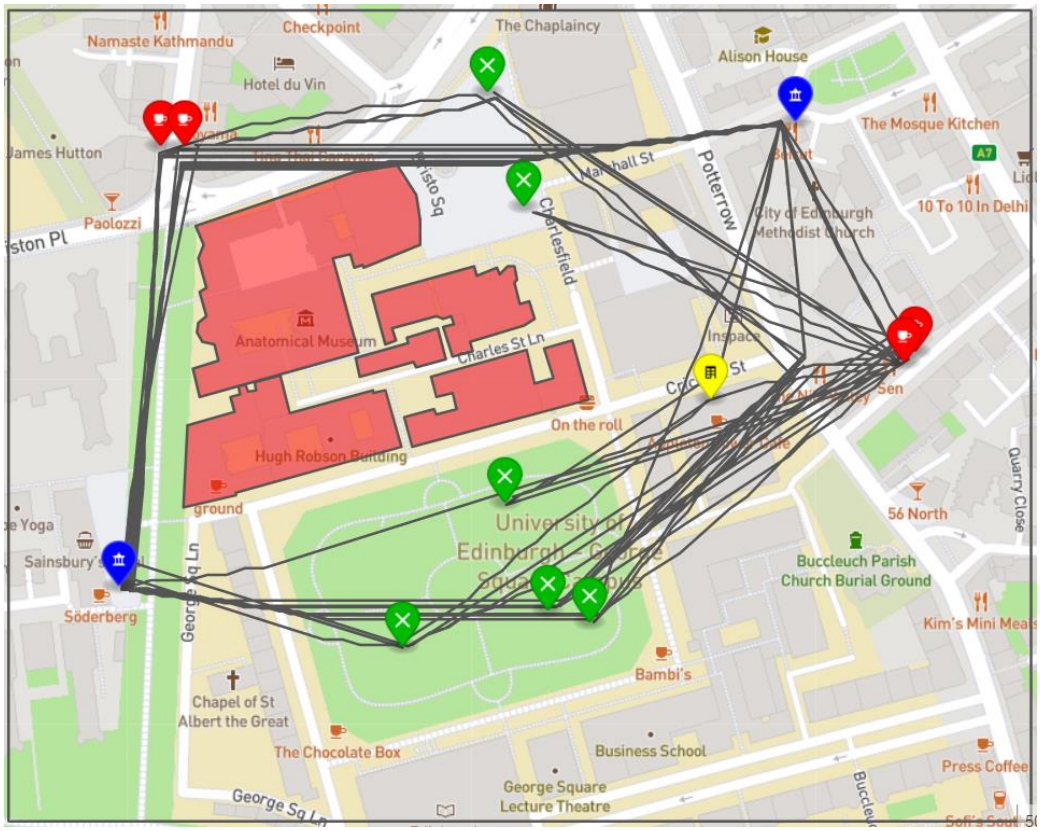
Every time we calculate the best angle, we have to be careful about our drone making an unexpected visit to a no-fly-zone.

In the above picture, if we were to travel from point A to point B, the green line shows us the direct route (straight line) that gets created and considered when determining whether we need to divert to landmarks or not. However sometimes, due to the fact that this straight line can be going towards any angle and that our drone can only move towards angles of multiple 10, this can cause the drone to make an unexpected visit to a no-fly-zone. The blue line shows us the drone's path, and we can see that it does indeed intersect with a no-fly-zone at the start. So, before every move, when we calculate the angle, we must check for line intersection with any no-fly-zone, and if there is an intersection, we adjust the angle +10 or -10 (we choose the one that does not make us intersect with a no-fly-zone, or if they both do not intersect, then we choose the one that gets us closer to the destination). If these angles still intersect, we keep checking until we find a valid angle.

Here is a flowchart for what the algorithm does **for each destination**:



Here is the rendered flightpath of my drone on 19/12/2023.
Percentage monetary value is 90%.



Here is the rendered flightpath of my drone on 10/06/2023.
Percentage monetary value is 100%.