1. **Software Architecture Description**

My application requires 10 classes in total. The main class is App.java, and the other classes are responsible for connecting the server, moving the drone, reading the map and get the fly path etc. Here are the classes I used:

App – This is the main class which is responsible for running the drone application and it would require calls from other classes and methods. It also would generate GeoJson file of flightpath and two database tables which are “deliveries” and “flightpath”. In the main function, this method sets up everything required for the drone’s flight, such connect to servers, initialize the map, start the flight, and generate the files that we need.

Drone - This is required so we can control the drone. The drone will be used in many functions next, so it is neater to keep all its functionality in its own class. It also means in the future more than one drone could be controlled at the same time within the app. For example, it contains nextMove(), which simulate how the drone move in the map.

Drone\_Map - This is required to specify what the drone will be navigating and create the visuals of drone flights. Right now, it creates the confinement area, no-fly zones and sensors. The constructor creates an object with the infoReader for the desired server, and the date of the flight requested by the user.

HttpConnection - This is required to create a connection to the http server of our choice. It is a utility class which will be called in the infoReader class.

SQLConnection – This is required to create a connection to the SQL database. It is a utility class which will be called in the infoReader class.

infoReader - This is used to read specific files from servers like HTTP and SQL database into their respective classes. It is also used to read the content from webs server and the database. It contains three methods, readBuidings(), readMap() and readLocations().

Location - This class is used to keep track of, and control, the drone’s locations. In my implementation it is used for the drone and all the shop’s locations. To be specific, it contains the algorithm of the drone’s moving, and how the bearing changes if the next move is invalid.

LocationUtils – This class is required to parse the What3Words location Jsons to other classes within the application can access the information. Specifically, the file holds the location details for a selection of What3Words string. The JSON file is composed of a single object with a nested schema of the country, coordinates, and areas of the What3Words square, etc. The toString() method allows us to pull the information from objects.

Menu – This class is required to store the components. of shop’s information and make it convenient for other class to call the items or locations of shops.

Order – This is required to represent the Order in this virtual environment. It contains GenerateRoute(), updateRoute(), and delivered(), which is the status of the order.

1. **Drone Control Algorithm**

Here I will discuss the algorithm used in this system which the drone visits the shops and other logical decisions during the drone delivering orders.

* 1. **Determining the route of the drone**

I implemented a Greedy Best-First algorithm to find a the most efficient way to collect items from shops and deliver them to different places. Greedy simply finds the closest node, removes it from the list of nodes to visits, then finds the closest to that one. Algorithm 1 is the pseudocode of the Greedy algorithm implemented:

**GREEDY**()

1: list of shops points copy ← list of shops points

2: route ← {}

3: next shop ← min dist (starting location, list of shops copy start location)

4: route ← route in index of (next shop, list of shops points)

5: **for** each shop in list of shops points **do:**

6: next shops ← min dist (next shop deliverTo, list of shops copy)

7: route ← route in index of (next shop, list of shops points)

8: list of shops copy ← list of shops copy − next shop

9: **end loop**

10: **return** route

This method proved it worked in general and the flight path was successfully generated for the date that we were required to submit. It could satisfy all requirements of the system. However, the sampled average percentage monetary value might be small as it has an obvious drawback when the drone is blocked by the no fly zones. Our algorithm will force it to move straightly, and it will show a zig-zag flight path and many moves are wasted here. To improve, the algorithm of avoiding no-fly zones and confinement zones should be fixed (see next section). More comprehensive search algorithms, such as A\*, where a heuristic is used and an overall cost to the goal is taken in consideration, would likely prove much more efficient than Greedy.

* 1. **Avoiding No-Fly Zones and Leaving the Confinement Area**

To prevent the drone from flying into, or over, the areas like no-fly zones or flying out-of-bounds, I checked that the predicted journey of the drone from the current position of to the next did not intersect any of the boundaries. If it did, it must recalculate the bearing at which to travel in and check the move again.

Here is the persuade code of the algorithm.

**IS\_VALID\_MOVE ()**

1: proposed move ← create line (coordinates (current pos), coordinates (next pos))

2: cross barrier ← false

3: **for** each vertex in list of area vertices **do**

4: area barrier ← create line (coordinates (vertex), coordinates (next vertex))

5:  **if** intersect (proposed move, area barrier) then cross barrier ← true

6: **end if**

7: **end for**

8: **return** cross barrier

* 1. **Recalculate the Bearing and Flight Path**

If the bearing is required to recalculated, the bearing will add and subtract 10 degrees each time, until the next move is valid. If it goes back the previous position, it will change the direction (add 180 degrees).

Here is the demonstration of how the algorithm is applied.

**NEW-BEARING()**

1: increment bearing ← false

2: counter ← 1

3: **while** not a VALID-DRONE-MOVE() **do**

4: **if** increment bearing then

5: bearing ← bearing + 10 ∗ counter

6: **if** difference (bearing, lastvalidbearing) =180 then bearing←bearing − 10

7: **end if**

8: increment bearing ← false

9: **else**

10: bearing ← bearing − 10 ∗ counter

11: if difference (bearing, lastvalidbearing) =180 then bearing←bearing + 10

12 **end if**

13: increment bearing ← true

14: **end if**

15: counter + +

16: **end while**

2.4. **Return the Start Point**

The max number of move of the drone is 1500, and the drone will try to finish all delivery orders before it returns. The drone will fly back to the start point if the number of the rest moves only allows the drone fly back. If this situation is detected, the drone will return.

1. **Results**

The results of flight path are not perfect as the algorithm still can be improved. I choose 15/09/2022 and 16/09/2022 as examples. Both testing data has 1500 moves and returns the start point finally. The results of GeoJson files are displayed here:

**Drone-15-09-2022**

**Drone-16-09-2022**

**绿色的地图

描述已自动生成**