Informatics Large Practical: Coursework 2 Report

Oliver Pan s2103701

Part 1: Software Architecture Description

1.1 Java Classes

App: the main class where the application receives command from command line, and creates necessary objects from existing classes to read order, make the drone move and record its motion, and finally generates required json and geojson files.

CentralArea: the class following singleton pattern and containing method of deriving corner coordinates of central area data from REST server and storing it. The class encapsulates central area information in an instance which can be called by other classes when necessary.

NoFlyZone: the class containing methods of parsing no fly zone information from REST service and using it to construct and store no fly zones. The class encapsulates no fly zone information which can be directly used by other classes when necessary.

Menu: the class defining a food item object, including fields of its name and price in pence.

Restaurant: the class representing a restaurant with its essential fields. It contains method of deriving restaurant details from REST server.

OrderOutcome: the enum including all possible outcomes for a given order based on details of an order. For a valid order, it denotes whether it has been delivered by the drone. For an invalid order, it denotes which section of the order is invalid.

Order: the class representing an order. It also contains helper methods deriving all orders of a given day from REST server and checking their validity and outputting details of valid orders in the order of the distance between their corresponding restaurant and starting position of the drone.

Direction: the enum containing 16 directions that the drone can move in each step and their corresponding angles, with 0 degrees representing east and 90 degrees representing north. This class is used to define what a single move of the drone can be.

LngLat: a class defining the coordinate of any point on the map with longitude and latitude. It can represent the position of the drone, or the location of points in central area or no fly zone. The class also contains methods for the position of the drone.

FlightPath: a class that contains algorithms to plan and save the shortest path for the drone to reach a specific restaurant and back and time cost for each step, provided with information of no fly zones and central area. This class is essential for implementing the drone's shortest path finding function.

Drone: the class for initialising object of drone with its initial battery and starting coordinate, and calls method in Order to obtain valid orders of a given day and method in FlightPath to plan the overall route for orders on that day. This is the core class for integrating order reading function and route planning function to implement the drone's delivery for a given day.

JsonOrder: a class for initializing object that records required information of an order in a day for the purpose of writing deliveries json file. This class is created because directly writing fields of Order object does not meet the requirement of deliveries file.

JsonRoute: a class for initializing object that records routes for delivering an order in a day and writing them into flightpath json file. This class is created to reuse the generated routes by Drone class and formulate them the way the flightpath json file specifies.

1.2 Application Pipeline

After providing App class with URL and specific date to access the REST service in command line, App will initialise a Drone object with battery of 2000 and starting position of Appleton Tower coordinate. On the other hand, the NoFlyZone and CentralArea class will use the given URL in addition with the no fly zone suffix and central area suffix respectively to parse date from REST server using jackson APIs, and construct no fly zones using Line2D class APIs and central area, which will become fields of the Drone object. Also, the Restaurant object will use the URL appended with the restaurant suffix to fetch restaurant coordinates and menus using jackson. The static method in Order will first use the URL appended with the order suffix and date to derive information of orders on that day and parse them as a list of Order objects.

With details of each order, the validity of each order is checked by verifying each of their fields. Concretely, the cvv number should be 3 digits and card expiry month

should be no later than the month inputted in command line. The Luhn's algorithm, an algorithm used to check the validity of credit card number, is implemented to check the validity of the credit card number. Next, helper function validRestaurantItems() will be used to check the validity of the restaurant and food items in each order, where a list of Restaurant objects will be used to check if all items in an order come from the same restaurant. Names of all items should all come from the corresponding restaurant of the order and the total number of them should not exceed 4. Only when an order satisfies all these requirements will it be considered valid. Valid orders will be labelled with OrderOutcome ValidButNotDelivered, and their corresponding restaurant as well as its coordinate will be saved, while invalid orders will be labelled with OrderOutcome specifying which incorrect part makes them invalid. Since the performance of the drone is assessed by the number of orders it deliver on a single day, therefore we expect orders that have restaurants with shorter distance from Appleton Tower to be delivered first, and Collections will be imported to sort these orders according to this requirement and place them into an array list called validOrders in Drone object. Orders whose restaurants have shorter distance from Appleton Tower will be placed at the front of the list. Then all invalid orders will be added after sorted valid orders.

For each valid order, Drone object will generate a FlightPath object and use the implemented method aStarPathFinding() to find the shortest round trip path to its restaurant. An array list of LngLat objects called route will be returned by the method to denote each coordinate of the shortest path of the order and another array list of Double objects called angles will be used to store the direction of each step. An array list containing Long objects called times will record time cost for computing the coordinate of each step. If the number of steps for this entire path plus 2 (the battery consumption for hovering when picking food in restaurant and delivering food in Appleton Tower) is less than or equal to remaining battery of the drone, the drone's remaining battery will be deducted by that length plus 2. Also, route, angles and times for this order will be respectively added as values of hash maps recording the entire route, directions and computation time of each step for that order in Drone object, and that order's status will be replaced with Delivered. However, if the condition is not met, computed route and angles will not be added, and Drone will stop computing path for remaining orders, since they need the same or more steps for steps than this order.

Finally, array list in Drone recording each order's outcome will be used and converted to JsonOrder objects to output deliveries json file, and hash maps with order number of each delivered order as key and recording route, angles and times on that day will be converted to JsonRoute objects in order to output flightpath json file. The route list will be used to output the required geojson file.

Part 2: Drone Control Algorithm

2.1 Path-finding for Each Order

The algorithm I use to implement drone's route planning function on any given order is a* algorithm. The main idea of this algorithm is to start from drone's starting point, check all of its 16 directions in turn and calculate their heuristic values, make a move towards the direction with lowest heuristic value and repeat previous steps for the position reached after movement, until reaching the destination. Here, starting point refer to the Appleton Tower position, and the destination is a point within 0.00015 degrees distance of the target restaurant.

Concretely, the heuristic in my algorithm for each position, denoted as f, is composed of two parts: g and h, where g is the total cost from starting point to current position and must be a multiple of 0.00015, while h is the estimated cost from current position to the destination. I compute h by calculating the euclidean distance between current position and destination. Finally, f is the sum of g and h.

Besides, I create two array lists named openTable and closeTable respectively to help with the iteration of checking process of each position. When the current position is computing the f value of each direction, an empty openTable will be created, and it adds all next positions with the f value after making a move towards their corresponding directions and sorts them based on f, and the next position with lowest f will be the first element of openTable. The closeTable is created to record points whose 16 directions have all been computed with f values. If a point is in closeTable, a* algorithm will no longer check it. A special case of the algorithm would be a drone's next position is within closeTable. If this happens, the algorithm will compare the f values of the position in closeTable and that as the drone's next position. If the point as the drone's next position has lower f value, the f value of that point in closeTable will be updated and its previous position will be updated as drone's current position.

To record the path, I innitialise a stack called pathStack. It will first store the final point computed by the algorithm and keep track of its previous position until reaching the starting position and add all of them.

The pseudocode of a* is shown below:

```
Algorithm a*(LngLat destination)
    Initialise an empty array list openTable
    Initialise closeTable
    Drone.startPosition.g <- 0
    LngLat currentPosition <- Drone.startPosition
    currentPosition.previousPosition <- null
    Initialise LngLat finalPosition
```

```
boolean closeToDestination <- false

while(! closeToDestination)
    clear openTable
    for all 16 directions d do
        LngLat nextPosition <- currentPosition.nextPosition(d)
        if (nextPosition.closeTo(destination)) then
            finalPosition <- nextPosition
            finalPosition.previousPosition <- currentPosition
            closeToDestination <- true

        else
            nextPosition.previousPosition <- currentPosition
            nextPosition.g <- nextPosition.previousPosition.g + 0.00015
            nextPosition.h <- nextPosition.distanceTo(destination)
            nextPosition.f <- nextPosition.g + nextPosition.h
            openTable.add(nextPosition)

        if closeTable.contains(nextPosition) then
            int i <- closeTable.indexOf(nextPosition)
            LngLat temp <- closeTable.get(i)

            if temp.f > nextPosition.f then
                temp.f <- nextPosition.f
                temp.previousPosition <- currentPosition

    closeTable.add(currentPosition)
    sort openTable based on f and arrange elements in ascending order of f

    LngLat previousPoint <- currentPosition
    currentPosition <- openTable.get(0)
    currentPosition.previousPosition <- previousPoint

Initialise Stack pathStack
LngLat node <- finalPosition
while(node.previousPosition != null)
    pathStack.push(node)
    node <- node.previousPosition
return pathStack
```

To obtain the way back to starting point, I only need to reverse the shortest path from Appleton Tower to the restaurant, and the direction of each step for the reversed path can be obtained by adding 180 degrees to each direction in direction list from Appleton Tower to the restaurant and modulating it with 360 degrees.

2.2 No Fly Zone Avoidance and Central Area Requirement Fulfillment

When the drone determines whether its next move will enter a no fly zone, I first iterate through each no fly zone object and use contains() method of the Line2D class to determine if drone's next position will enter a no fly zone. However, this can be computationally expensive due to the high precision of coordinates and long computation time for determining if a point is within a polygon area. To address this problem, I design another heuristic to reduce the computation time.

I create a method called magnify() in LngLat. It multiplies the lng and lat of every LngLat object by 100000 and round them to integers. This is to avoid the complexity of high precision double computation and 100000 is a good scale factor for those coordinates. We will apply this method on every edge points of no fly zones and add them to an array list field called noFlyZoneEdgePoints in NoFlyZone class. Next, on every edge of no fly zones, from the starting point of the edge, calculate the coordinate of a point every 0.00005 degrees, before going over the end point of the edge. It should be noted that the starting point and end point are two consecutive coordinates of a no fly zone edge in the REST server. Then apply magnify() on those calculated points and add them to noFlyZoneEdgePoints. The aim of this step is to construct an approximate contour of each no fly zone using discrete number of points. The reason I choose 0.00005 for the interval of two edge points is because some edges are short, and 0.00005 can ensure there is at least one point on the edge other than starting and end point of that edge to the greatest extent. The pseudocode for this step is shown below:

```
Algorithm magnify(LngLat point)
    int updated_lng <- round_to_integer(point.lng * 100000)
    int updated_lat <- round_to_integer(point.lat * 100000)
    return new LngLat(updated_lng, updated_lat)

Algorithm construct_no_fly_zone_contour (NoFlyZone nfz)
    Initialise ArrayList noFlyZonePoints
    for all LngLat edgePoints do
        noFlyZonePoints.add(magnify(edgePoints))

    for all Line2D edge in nfz do
        LngLat startingPoint <- edge's first element
        LngLat endPoint <- edge's second element
        angle <- arctan(startingPoint.lat - endPoint.lat, startingPoint.lng -
    endPoint.lng)
        i <- 1
        while (0.00005 * i < edge.length) do
            noFlyZonePoints.add(magnify(startingPoint.lng + 0.00005 *
        cos(angle), startingPoint.lat + 0.00005 * sin(angle))
```

```
                i <- i + 1
        return noFlyZonePoints
```

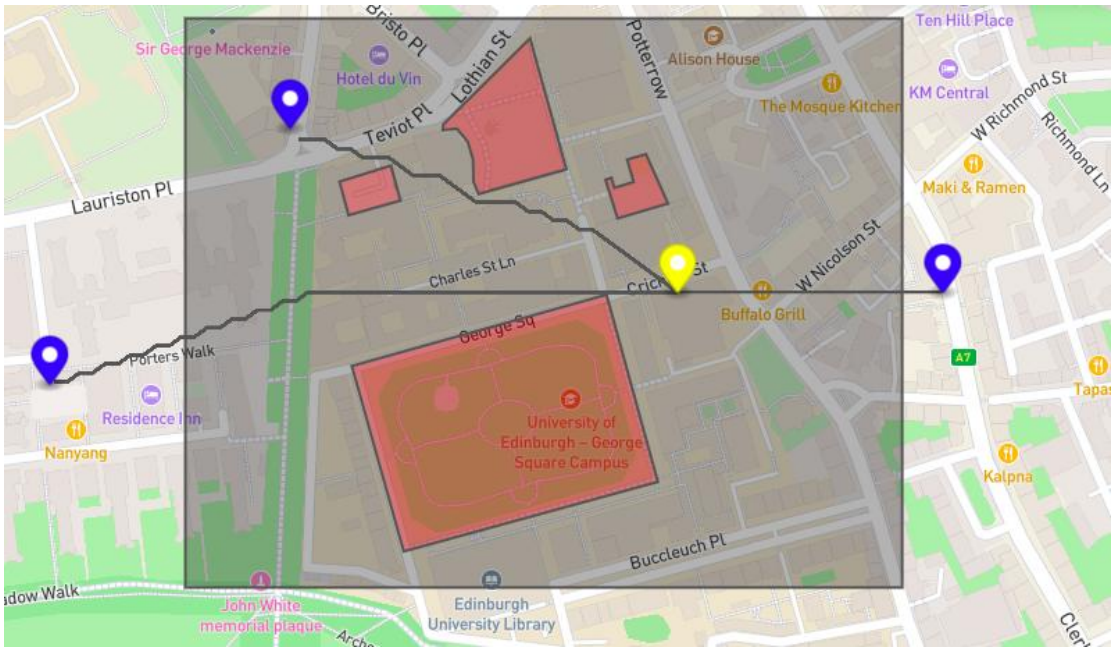
Then, a closeToNoFlyZone() method is implemented to determine whether the next position towards the a specific direction will enter a no fly zone. It will first apply magnify() on the next position coordinate of the drone, and loop through noFlyZoneEdgePoints in all NoFlyZone objects and find the edge point closest to it, then calculate the distance between next position and that point. If that distance is less than a threshold value, that direction will be discarded since the drone will be highly likely to enter a no fly zone. On the other hand, the calculation of distance between magnified integer coordinates and discrete number of comparisons of distance between drone's next position and edge points will not be computationally expensive. The threshold value I choose after many attempts is $\sqrt{10}$. If the path from Appleton tower to restaurant does not intersect with any central area, then the path back to Appleton Tower will not enter central areas either.

To avoid the drone from first leaving the central area and then going back into it before reaching the restaurant, we will add another condition to our a* path-finding algorithm. Given that the current position is not the starting point and it is not inside central area and its previous position is inside the central area, then when iterating through directions of the current position, a direction will not considered if the position's next position after moving towards this direction is inside central area. On the other hand, this also helps ensure that the drone will not go outside the central area again if it has entered central area before finishing delivering order.

2.3 Flight Path Rendering

To test my application performance, I choose to parse orders on April 15th,2023 and plan routes for them. The results are shown below:

There are 40 out of 47 valid orders on that day. 29 orders have been delivered before the drone runs out of battery, among which 10 correspond to Domino's Pizza - Edinburgh - Southside, 10 correspond to Civerinos Slice, 9 correspond to Sodeberg Pavillion. The number of steps for shortest round trip to Domino's Pizza - Edinburgh - Southside is 40. The number of steps for shortest round trip to Civerinos Slice is 60. The number of steps for shortest round trip to Sodeberg Pavillion is 96. After the finishing deliveries of the day, the drone's remaining battery is 78 out of 2000. The flight paths in drone-2023-04-15.geojson are shown below:



It should be noted that Sora Lella Vegan Restaurant is not displayed in the graph since the drone has not delivered any order from it. If the drone would be able to deliver an order from that restaurant, the number of steps of shortest round trip to it would be 210.

It should also be noted that the drone's flight path to Sodeberg Pavillion does not intersect with the George Square no fly zone, even though in the graph above it looks like they intersect. The magnified graph is shown below: