Informatics Large Practical: Coursework 2 Report

Zihao Cai S1915409

Part 1: Software Architecture Description

1.1 Java Classes: why they exist

App: the main class. It runs the application by accepting data, constructing objects, moving the drone, and outputting the required tables and geojson file.

ReadWeb: This class keeps all methods to connect to and parse files from the web server. So that other classes calls it if web server's content is needed. Such design provides encapsulations and removes unnecessary repetitions. Future modifications are also straight forward if the content on the web being changed.

ReadDataBase: Similar to ReadWeb, it connects to the database and fetch required content from it. It also creates tables named flightpath and deliveries in the database as required. So the connection to database is only required once when calling the class's constructor.

Map: semantic object representing the map the drone is traversing. Objects on the map, such as no-fly-zones, landmarks and locations of orders are created by this class.

Drone: semantic object describing the drone. It calculates the path of the drone to deliver an order and move the drone to do so. Since it represents a single drone, it is easy to support the case if the delivery system could support multiple drones in the future.

LongLat: semantic object expressing the longitude and latitude position of an object: drone, landmarks, points of no-fly-zones, shops, and delivery point. It also contains helper functions for the drone.

Menus: stores the items available in all shops. Since only the price and shop's location of an object is needed. The class disregard other provided information and only keeps those two data as HashMap. However, if more information needed in the future, it is easy to bring them back.

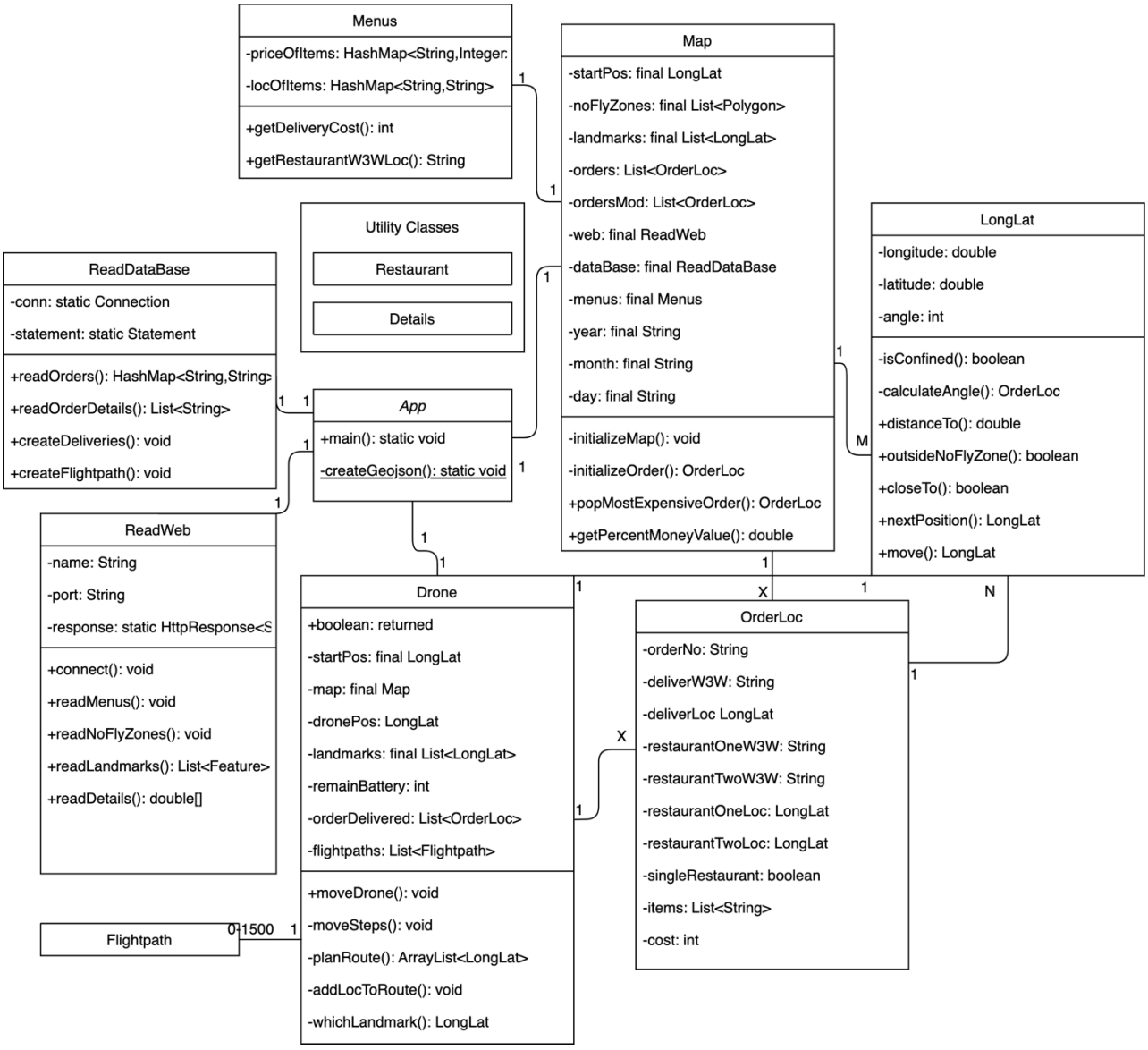
OrderLoc: class representing an order's locations and other useful information. Since the drone processes deliveries orders by orders, it is straightforward to store both shops and delivery location of an order into one OrderLoc object.

Flightpath: class storing one step of the flightpath made by the drone, as required by one row of the flightpath table the app created in the database.

Details: class representing the Details.json file of a word3word location in the web server. This class is used when parse data from the web server.

Restaurant: Similar to Details, representing one instance in menus.json file (menu of a shop / restaurant) in the web server. Also used when parse data from web server.

1.2 UML Class Diagram of the Application



1.3: Application Pipeline

After calling the App class via command line, the class creates one ReadWeb and one ReadDataBase instance to connect to servers. It then initializes the scene by creating one Map and one Drone instance respectively.

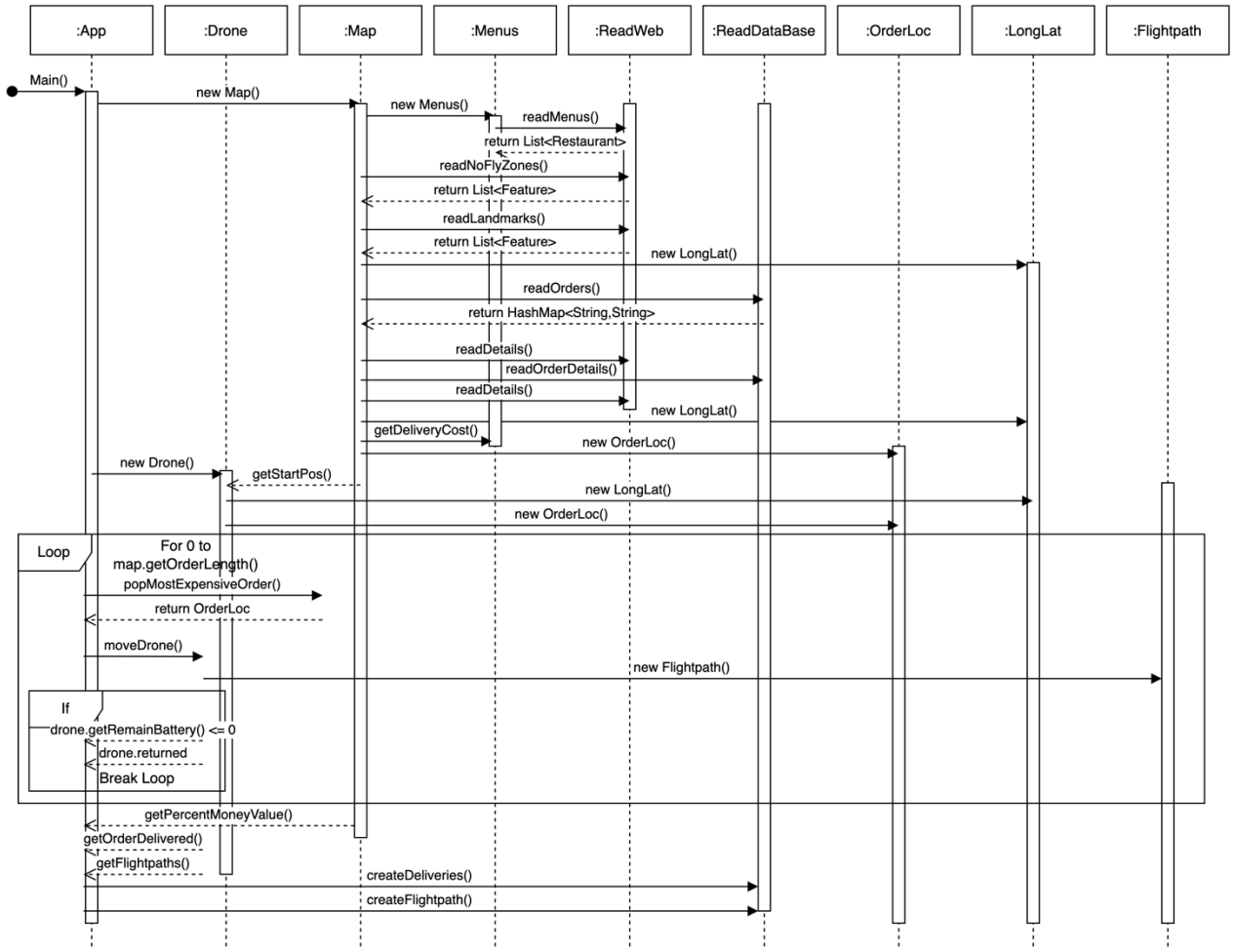
The Map instance uses the ReadWeb instance defined earlier to fetch no-fly-zones and landmarks. It also creates a Menus entry that calls the same ReadWeb instance and fetches required information regarding to items. The Map instance then creates multiple OrderLoc instances — storing shops and delivery locations and other useful information such as orderNo and total cost — by fetching data using the previous ReadWeb and ReadDataBase instance. All positions, including landmarks and shops & delivery positions, are logged into LongLat objects.

Then the App class starts to move the drone for today's delivery order by order: it keeps feeding the most valuable remaining order to the Drone instance until the drone stops: returned to starting position after finishing delivery, returned but does not finish delivery, or ran out of battery mid-way.

After receiving the current order to deliver, i.e., the most valuable remaining order fed by App class, the Drone instance calculates the path to validly deliver this particular order. Then it moves until it stops: by the conditions mentioned as above. Meanwhile, it stores every move made by the drone into a list of Flightpath objects and all orders delivered. Such delivery process of the drone would be described more thoughtfully in the later "drone control algorithm" section.

After the drone stops moving, the App class records its flightpath and the orders delivered from the Drone instance. Then the App class calculates percentage monetary value of the day. Finally, the App class logs the flightpath into geojson file and delivered orders into tables in database by calling methods in the previous ReadDataBase instance.

1.4 UML Sequence Diagram of the Application



Part 2: Drone Control Algorithm

The three parts of the algorithm mentioned in this report is split into portions of multiple methods in my actual implementation. So although details of the pseudocode here in the report would be different from the code in the actual app, the underlying intuition would be the same.

2.1 Control the flight and back

As mentioned before, the drone process a day's delivery order by order: the App class repetitively feeds one order to the drone. Since an order can be made of items from no more than two shops and the drone needs to travel to the shops before it gets to the delivery location, a heuristic is used to determine which shop to go first if two shops are present in an order. The heuristic is simply the straight-line distance connecting current drone position and shop's position: the drone travels to the closer shop first, then the next shop, then the delivery position. Drawbacks exist since this algorithm does not support an order consists of items from more than two shops. If say, items from 10 shops are supported in one order in the future, such scenario could be treated as a TSP (traveling salesman problem) and more comprehensive search algorithm, such as two-opt or A star, will be required.

Pseudocode to determine the sequence to visit locations in an order:

```
Algorithm Plan_The_Route_For_Current_Order ( order, current_position )
1. Initialize List route
2. if order.have_only_one_shop then
3.      route.add( order.shop_position, order.delivery_position )
4. else then
5.      if distance_of ( current_position, order.first_shop_position ) <
6.              distance_of ( current_position, order.second_shop_position ) then
7.              route.add ( order.first_shop_position, order.second_shop_position,
8.                      order.delivery_position )
9.      else then
10.     route.add ( order.second_shop_position, order.first_shop_position,
11.             order.delivery_position )
12.     return route
```

To make one single move, the drone calculates the angle from its current position to the desired position. Then this angle is transformed into a valid bearing of the drone by transforming it to the drone's coordinate system and rounding it to multiple of ten. Afterwards, the drone flies a straight line of length 0.00015 degrees in that angle.

When the drone receives the last order of the day from the App class, it adds its starting location to the end of its route: after reaching shops and delivery position of the last order, as mentioned earlier. The drone would terminate the flight when it delivered the last order and reached the starting position. Moreover, the drone also prepared for the case when it does not have enough battery left to deliver the current order. After it calculated the path for the current order and before moving, the drone calculates another heuristic: total amount of battery required to travel in a line connecting all points in the path. If such heuristic is larger than the drone's remaining battery, it gives up the current order and tries to return to starting position immediately. The drone would always attempt to go back to initial position regardless of whether it could deliver all orders or not. But there exists a possibility that the drone runs out of battery when returning or when delivering — the heuristic is simply an estimation after all.

2.2 Avoiding no fly zones

Generally, the drone utilizes landmarks to avoid no-fly-zones. If a straight line connecting two points — a heuristic connecting a shop and delivery point for instance — intersects with any boundary line segments of the no-fly-zones, the drone would first move to a landmark, then it flies from the landmark to the desired position.

Pseudocode to determine if moving a straight line from current position to a desired position crosses no-fly-zones:

```
Algorithm Check_If_Path_Is_Outside_No_Fly_Zone( current_position, desired_position )
1. path <- line ( current_position, desired_position )
2. for all no_fly_zone do
3.      for vertex in vertices_of_this_no_fly_zone do
4.              no_fly_zone_edge <- line ( current_vertex, next_vertex )
5.              if intersect ( path, no_fly_zone_edge ) then
6.                      return false
7. return true
```

To determine the landmark being used, the algorithm first gets all landmarks unblocked by no-fly-zones from current position and selects the landmark that minimize total distance -- from current position to landmark, then to desired position. If all landmarks being blocked by no-fly-zones, the algorithm selects the minimum-distance-landmark from all landmarks. Thus although the current configuration have two landmarks, the algorithm could handle more if the map being changed in the future. Since the drone hovers when it is close to any point to deal with its payload, drawback exists when the drone hovers at a landmark: wasting the drone's battery. However, as we will see later, this problem could be neglected since the drone have plenty of battery left even in the most heavy-duty day of 2022. In December 2023, with 27 orders a day, only about 2% of battery is wasted.

Pseudocode to determine the landmark being used when moving to desired position directly is impossible:

```
Algorithm Determine_Which_Landmark_To_Use ( previous_position, desired_position )

1. Initialize List acceptable_landmarks

2. for all landmarks lm do

3.      if Check_If_Path_Is_Outside_No_Fly_Zone( previous_position, lm ) and

4.              Check_If_Path_Is_Outside_No_Fly_Zone( lm, desired_position ) then

5.      acceptable_landmarks.add (lm)

6. if size_of (acceptable_landmarks) == 0 then

7.      acceptable_landmarks <- landmarks

8. min_distance <- infinity, min_landmark <- acceptable_landmarks.first_element()

9. for all acceptable_landmarks relm do

10.     distance <- distance_of ( previous_position, relm ) + distance_of ( relm,
desired_position )

11.     if distance < min_distance then

12.             min_distance <- distance, min_landmark <- relm

13. return min_landmark
```

To avoid no-fly-zones locally in a single step of 0.00015 degrees, the drone recalculates its bearing if the step crosses no-fly-zones or being outside the drone confinement area. The drone does so by alternating the bearing by +10, then -10, +20, -20, +30… degrees until the move is valid. Also, moving back to the previous location is forbidden since it would cause the drone to get stuck in a loop.

Pseudocode to move a valid single step of 0.00015 degrees:

```
Algorithm Move_A_Valid_Step( current_position, next_position, bearing, pre_bearing )

1. adjustment <- 10

2. while not Check_If_Path_Is_Outside_No_Fly_Zone( current_position, next_position )

3.      or not Inside_Drone_Confinement_Area(current_position, next_position ) do

4.      bearing <- bearing + adjustment

5.     if abs(bearing - pre_bearing) == 180 then

6.           bearing <- bearing + 10*adjustment/abs(adjustment)

7.      next_position <- attempt_to_move (current_position, bearing)

8.      adjustment <- -(adjustment +  10*adjustment/abs(adjustment))

9. return bearing
```

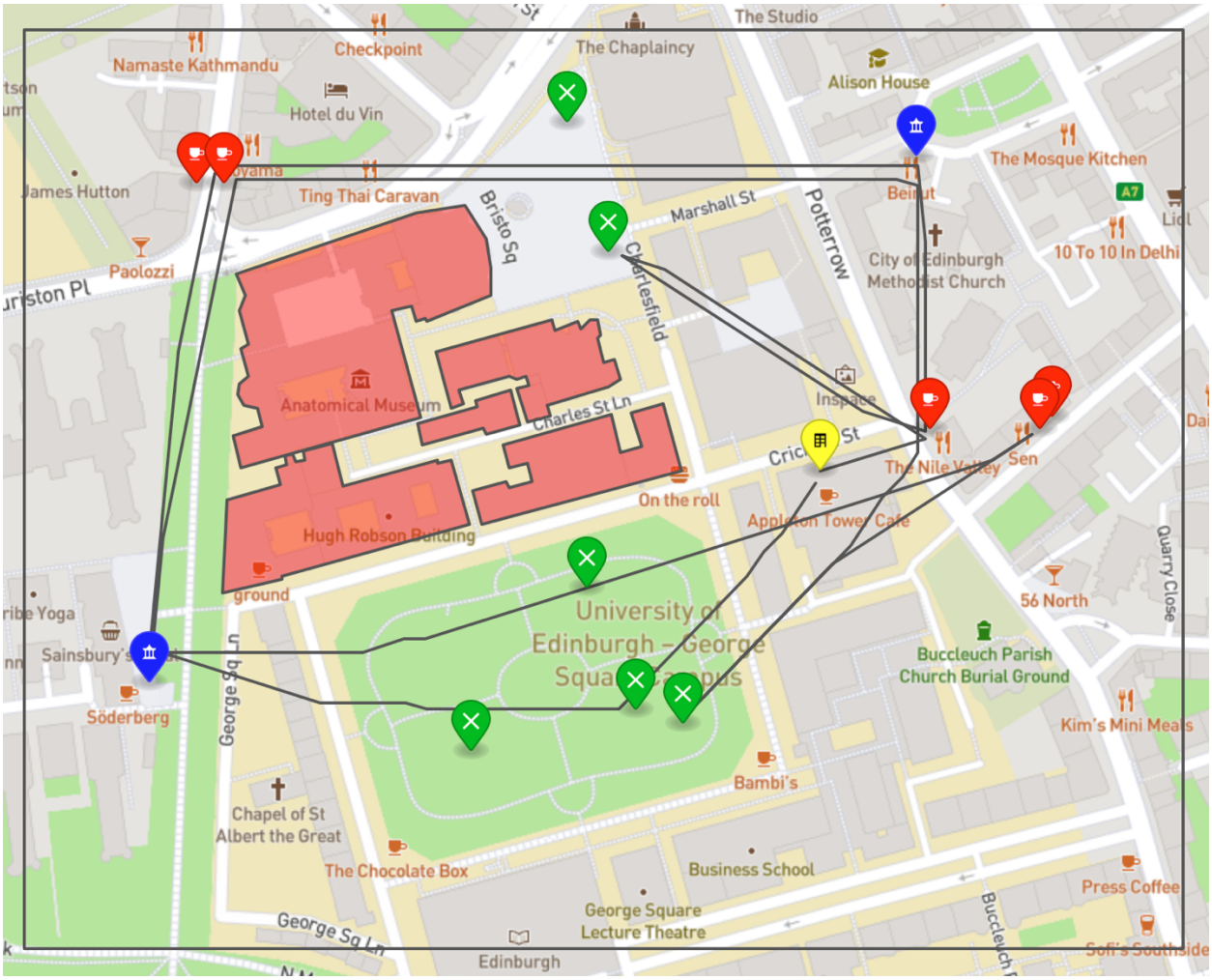
2.3 Maximize drone score

A greedy-based-approach is used to maximize the score. Since the App class moves the drone orders by orders, it gets the most expensive order among the remaining ones and call the drone to deliver this order. A simple "get the max element in a list" approach, similar to the one in `Check_If_Path_Is_Outside_No_Fly_Zone()`, is used to determine the most expensive remaining order. At first, I planned to choose the order that requires the least amount of moves if there are multiple orders with equivalent monetary value. But soon I observed that most order values are different, so such approach is unnecessary.

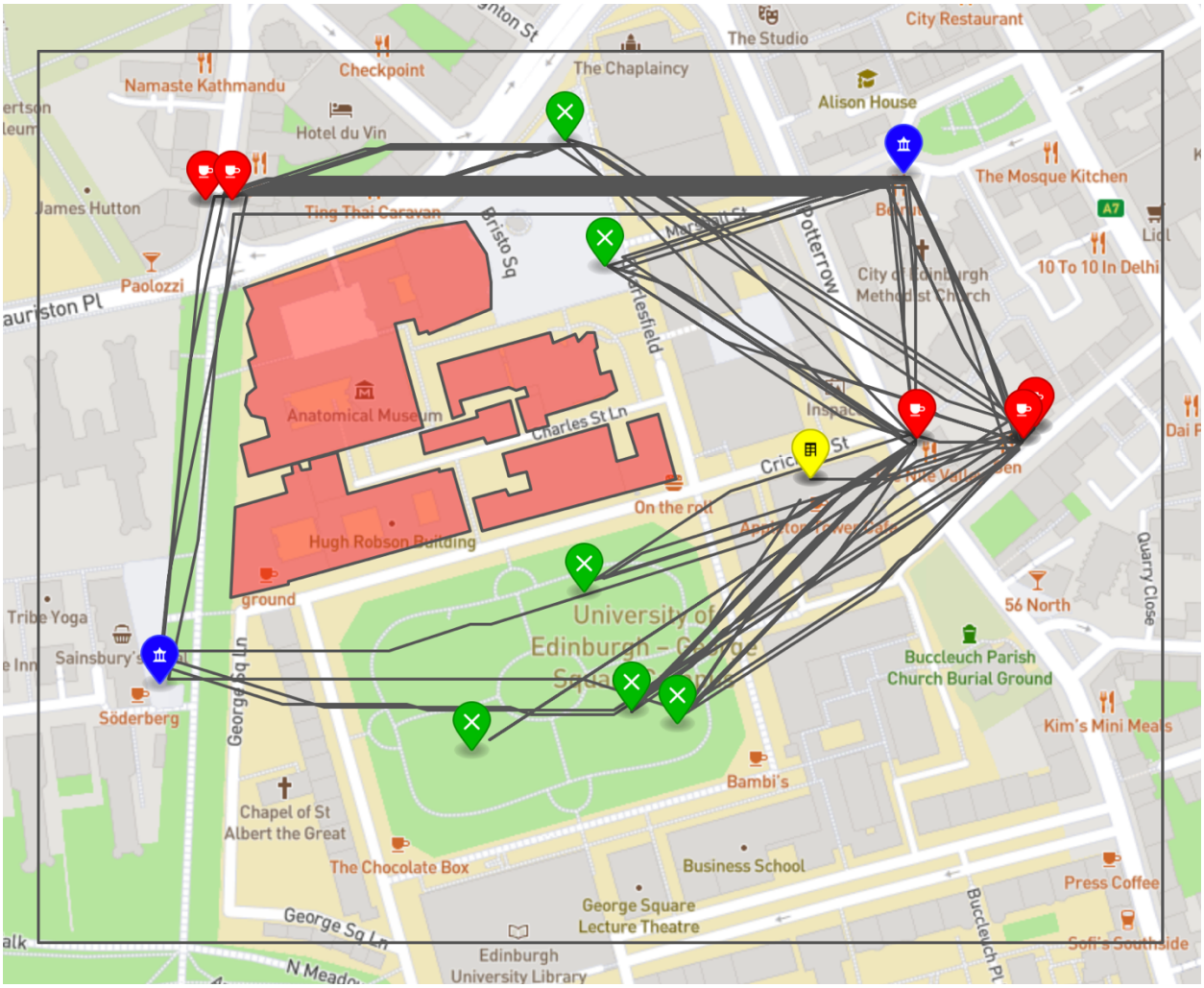
After extensive testing, in year 2022, the percentage monetary value metric of a day strictly equals to 1.0 and the sampled average percentage monetary value also strictly equals to 1.0. Meaning the drone could confidently deliver all orders of 2022, even in December when there are a total of 15 orders a day. In fact, the drone still has approximately 400 moves left after it finished delivery on the most challenging days of 2022 and returned to starting position. In December 2023, when there are 27 orders in a day, the sample average percentage monetary value is 0.93 by taking a random sample of 7 days in December 2023. Such results show the algorithms is effective enough in the current scenario.

2.4 Flightpath Renderings

Flightpath of 03-01-2022, the most light-duty month with 4 orders a day. The percentage monetary value is 1.0 and there is 1239/1500 remaining battery left after delivery and returned. Note that some delivery positions (green marker) are not used. Since the amount of order required to deliver is small and the flightpath is sparse, this figure provides a clear visualization of the flightpath when delivering a single order. It demonstrates how the drone moves between two locations, utilizes landmarks to avoid no-fly-zones, and returns to starting position after finishing delivery.



Flightpath of 03-12-2023, the most heavy-duty month with 27 orders a day. The drone managed to deliver 24/27 orders and achieved a percentage monetary value of 0.958. The drone managed to return to starting position when it calculates that it could not finish the 25th order. After the drone returns, the remaining battery is 18/1500. This figure demonstrates how the drone delivers tens of orders. Since the desired paths are similar between two points, the flightpath clusters together when the drone travels between two locations over and over.



Reference

[1] https://google.github.io/styleguide/javaguide.html
[2] https://draw.io
[3] https://javadoc.io
[4] https://www.baeldung.com/java-jdbc
[5] https://www.tutorialspoint.com/