Informatics Large Practical Report

s2020153@Yinsicheng Jiang

November 30, 2022

Contents

1 Software Architecture Description

1.1 Structure Overview

The application is formed through a collection of 14 vital classes: App, CentralSingleton, CreditCard-Check, DeliveriesJson, Drone, FlightPathJson, GeoJson, Greedy, LngLat, Map, Menu, NoFlySingleton, Order and Restaurant. Together, they form an application that takes in a date, a server address, and a random seed **(mentioned in the coursework document, but not used in my program)** and outputs the result of an autonomous drone's flight for delivering pizza orders according to the request on the server.

The **App** is the main class that accepts the input of the user and returns the final results of all deliveries on a given day. It first checks the validation of the input offered by the user and if something invalid is caught, it will report the message of the fault and exit the system. After passing the validation check, it extracts various zone coordinates, restaurants, and orders from the server address by *getRestaurantsFromRestServer* method in the Restaurant class and *getOrdersFromServer* in the Order class. The *getOrdersFromServer* method is called in the drone constructor as the App creates a new drone instance. Afterwards, it starts delivering orders with the *deliver* method linked to the Drone class using the map of one order created as an instance of the Map class which contains the start and the destination restaurant and finally returns the order outcomes and flight paths in the JSON format together with one in GEOJSON for visualization by the methods in the corresponding JSON generating Class.

The **CentralSingleton** is a singleton class used to generate the central area on the map. It gets the coordinates in the LngLat form, which will be introduced later, of each vertex of the central area from the server by using *getCoordinates* method in this class and stores them as an array of LngLat for further utilization.

The **CreditCardCheck** is a class that helps check the validation of the credit card number used by the user for payment. It uses Luhn algorithm to check the prefix, total length, and "Mod 10" rule of the credit card number and decide whether the card number is valid or not. If the card number is set to invalid, the *InvalidCardNumber* exception will be caught and the system will print some error messages before exiting.

The **DeliveriesJson** is a class that helps generate the JSON object of delivery information for each order by using *generateJson* and *generateJsonArray*, including order outcomes, order numbers, and total costs.

The **Drone** is an important class that defines the control of the delivery process of the drone. It first and foremost extracts orders from the server, and it filters out invalid orders using the *order-sFilter* method in the Order class coupled with sorting orders by the distance to the destination restaurant from the Appleton Tower using the *sortOrders* method in the Order class. The Drone class has a method called *deliver* that starts delivering orders according to the drone's battery, using a search algorithm by calling the *start* method in the algorithm class Greedy. After finishing deliveries, the drone generates the results of all orders like flight paths and delivery conditions in JSON format to the result files. For flight paths, the drone only remembers the path from the start to the destination and follows its reverse version when going back which saves a lot of time. A way to improve efficiency is to remember all the paths so that the drone does not need to search the road again if the restaurant is the same as previous orders. But to avoid potential bugs, I did not implement this method.

The **FlightPathJson** is a class that helps generate the JSON object of flight paths for each order, including order numbers, from-longitude, from-latitude, angles, to-longitude, to-latitude and ticks since the start of the calculation.

The **GeoJson** is a class that helps generate the GEOJSON files of flight paths for visualization on geojson.io

The **Greedy** is a search algorithm class that will be introduced later in the algorithm section. Briefly, it uses Greedy-Best-First-Search to execute the flight path. Basically, calling the *start* method can start the iteration of the algorithm.

The **LngLat** is a class mainly dealing with coordinates. It is a representation of point coordinates in the form of (longitude, latitude). The distance, angles, and other properties which are related to coordinates are all calculated in this form. The areas like the central area and no-fly zones are represented as an array of LngLat. Inside this class, there is an *Enum* class called **Direction** that is defined as the legal moving direction of the drone. The inner *reverseAngle* method is used to find the opposite angle of the given angle.

The **Map** is a class that represents the map of each order. Its constructor contains all regions that should be shown on the map like the central area, no-fly zones, the starting point and the endpoint. But only the starting point and the endpoint are mostly used.

The **Menu** class stores the names and prices of pizzas in each restaurant.

The **NoFlySingleton** class is a singleton class used to generate the no-fly zones on the map. It gets the coordinates in the LngLat form of each vertex of the central area from the server by using the *getNoFlyZones* method in this class and stores them as an array of LngLat for further utilization. The format of coordinate systems represented in no-fly zones is different from that of the central area, so the method *transfer2LngLat* is used to change the format into LngLat for convenience.

The **Order** is a class that contains multiple functions help deal with orders of a given day. It has an inner static class Tuple that is defined as (restaurant, price) for later stored in the hashmap with the key to be the corresponding dish's name. Thereby, we can easily get the restaurant and its coordinate of the items in each order by using *getLngLatFromOrder*. For convenience, there is also a method called *invalidOrders* that stores the invalid orders and their error messages so that we can directly work on those valid orders and finally put all order outcomes together to generate the JSON file. In terms of filtering out invalid orders, methods *paymentCheck* and *pizzaCombinationCheck* are also implemented in this class for straining orders. If some orders are invalid, the exception message will be stored and eventually included inside JSON file. The inner Enum class **orderOutcomes** embodies all the outcomes that an order will result in.

The **Restaurant** class stores the name, coordinates, and menus of each restaurant. The method *getRestaurantsFromRestServer* helps download the restaurants from the server.

Overall, the *CentralSingleton, Drone, NoFlySingleton, Menu, Restaurant*, and *Order* are important elements and physical objects of the program. *Greedy* contains the main search algorithm for the drone. *LngLat and Map* are important for creating a smooth, logical, encapsulated program, and *CreditCardCheck, DeliveriesJson, FlightPathJson, and GeoJson* are used for checking validity and generating result JSON files. All these classes are important in constructing the structure of the program, ensuring the code is readable and easy to maintain.

1.2 Important Class Description

1.2.1 App Class

Methods

- *main()* is the main method for the whole program that receives the date, server address, and random seed typed in by the user. The class first checks whether the format of the date is YYYY-MM-DD. If it is not in this form, the program will return an error message like "Invalid date type!" and exit the system. After passing the check, it reads the information of Restaurants and orders from the server and creates a drone instance. Furthermore, it calls the *deliver* method in the Drone class to start delivering and gets the final results. Finally, it uses *generator* in each JSON helper class to convert the results into JSON format. In the meantime, if something unexpected happens, the class will catch it and give an error message before exiting the system.
- *isValidFormat(String format, String value, Locale locale): boolean* is the method that helps check the date format whether it is in YYYY-MM-DD or not. If it is not in that format, the method returns false, otherwise returns true.

1.2.2 CreditCardCheck Class

Methods

- *validitychk(long cnumber): boolean* checks whether the card number is valid or not. In this project, we only accept Visa-16 and MasterCard.
- *sumdoubleeven(long cnumber): int* & *sumodd(long number): int* are the methods help check if the card number follows the "Mod 10" rule that the sum of the total number must be divisible by 10. If so, return true. Otherwise, return false.
- *thesize(long d): int* returns the number of digits in d.
- *prefixmatch(long cnumber, int d): boolean* returns true if the digit d is a prefix for cnumber.
- *multiPrefixMatch(long cnumber, String cardType, int d): boolean* returns true if the prefix of cnumber matches d given the card type.
- *getprefx(long cnumber, int k): long* returns the first k number of digits from number. If the number of digits in number is less than k, return number.

1.2.3 Drone Class

Attributes

- *URL baseURL*
- *String dateTime*
- *Order[] orders*
- *ArrayList⟨Order⟩ filteredOrders*
- *HashMap⟨Order, String⟩ invalidOrders*

- *final int BATTERY = 2000*

<u>Methods</u>

- *deliver(ArrayList⟨Order⟩ orders): ArrayList⟨JsonArray⟩* starts delivering the filtered and sorted orders by starting the search algorithm and adding the outcomes, flight paths and all moves into different JsonArrays respectively coupled with the supervision of remaining battery. It finally returns an ArrayList that contains all three JsonArrays.
- *orderPath(Order order, Greedy greedy): ArrayList⟨Greedy.Node⟩* generates the flight path from the start point (Appleton Tower) to the endpoint (the restaurant of the order) by the Greedy-Best-First-Search algorithm.
- *finalPath(ArrayList⟨Greedy.Node⟩ path, Map map): ArrayList⟨Greedy.Node⟩* generates the final path by adding the reverse path to the former path returned by the search algorithm (*method orderPath*).
- *conditionPath(Order order, String condition): JsonObject* generates the JSON object for different drone conditions. If the condition is "power out", all the outcomes of remaining orders will be set to *ValidButNotDelivered* and stored to the JSON object. On the other hand, if the condition is "valid", the outcome of the order will be set to *Delivered*
- *findFlightPath(JsonArray flightPath, ArrayList⟨Greedy.Node⟩ steps, Order order): JsonArray* converts the coordinates into the flight-path format required by the JSON object.
- *pointsForGeoJson(JsonArray points, ArrayList⟨Greedy.Node⟩ steps)* converts the coordinates into the format required by GEOJSON.

1.2.4 LngLat Class

<u>Attributes</u>

- *double lng*
- *double lat*
- *String name*
- *final double TOLERANCE = 0.00015*
- *final double EPSILON = 1e-12*

<u>Methods</u>

- *isInPolygon(LngLat point, LngLat[] polygon): boolean* checks whether the object is inside or on the polygon.
- *inCentralArea(): boolean* checks whether the object is inside the central area.
- *intersectWithNoFlyZones(LngLat next): boolean* checks whether the next position the drone moves will intersect with no-fly zones. Return true if intersect.
- *distanceTo(LngLat lnglat): double* calculates the distance between the object and the target point.
- *closeTo(LngLat lngLat): boolean* checks if two points are closed enough.
- *nextPosition(Direction direction): LngLat* returns the next position after movement. If the drone hovers, the current position will be returned.

1.2.5 Order Class

Attributes

- *String orderNo*
- *String orderDate*
- *String customer*
- *String creditCardNumber*
- *String creditCardExpiry*
- *String cvv*
- *String priceTotalInPence*
- *String[] orderItems*

Inner enum

- **OrderOutcome, Tuple(String restaurant, int price)**

Methods

- *getDishes(Restaurant[] restaurants): HashMap*$\langle String, Tuple\rangle$ puts dishes of restaurants inside the hashmap for further reference.
- *pizzaCombinationCheck(Restaurant[] restaurants, String[] pizzas)* checks the pizza combination's validity of an order.
- *paymentCheck(Restaurant[] restaurants, Order order)* checks the validity of the payment.
- *getDeliveryCost(Restaurant[] restaurants, String[] pizzas): int* gets the total cost of the order.
- *getOrdersFromServer(URL baseURL, String date): Order[]* gets orders from the server.
- *getLngLatFromOrder(Order order, Restaurant[] restaurants): LngLat* gets the coordinates of the restaurant referred to the order items.
- *ordersFilter(Order[] orders, Restaurant[] restaurants): ArrayList*$\langle Order\rangle$ filters out the invalid orders returned by the two checks.
- *invalidOrders(Order[] orders, Restaurant[] restaurants): HashMap*$\langle Order, String\rangle$ stores the invalid orders as keys point to their error messages inside a hashmap.
- *sortOrders(ArrayList*$\langle Order\rangle$ *orders, Restaurant[] restaurants)* sorts orders according to the distance from AT to the restaurant.

1.2.6 Category Clarification

Map objects:

- **CentralSingleton, NoFlySingleton, LngLat, Map**

Delivery Operation Class:

- **App, Drone, Greedy, Menu, CreditCardCheck, Order, Restaurant**

JSON Generators:

- **DeliveriesJson, FlightPathJson, GeoJson**

2 Drone Control Algorithm

2.1 Search Algorithm

In real-life engineering, we do not need to care about extreme conditions like puzzle routes. So I implemented Greedy-Best-First-Search to efficiently search the path from the start to the destination. It basically tries to find the nearest way approaching to the destination during every selection of next movement. Here is the pseudo code.

Algorithm 1 Greedy-Best-First-Search(Map map)

```
1: if $map = null$ then return $null$
2: end if
3: $PriorityQueue\langle Node \rangle$ $openList.clear()$
4: $ArrayList\langle Node \rangle$ $closeList.clear()$
5: $openList.add(START)$
6: $ArrayList\langle Node \rangle$ $steps \leftarrow null$
7: while $!openList.isEmpty()$ do
8:     $current = openList.poll()$
9:     $closeList.add(current)$
10:    $steps.add(current)$
11:    $addNeighbourNode(current, map.end)$
12:    if $isPointNearEndInClosed(map.end)$ then return $steps$
13:    end if
14: end while
```

This method proved effective enough in general and was pretty efficient for the dates that we were required to submit. By utilizing the priority queue, we can always get the head of the queue as the best solution of the next movement. The possibilities of 16 directions will be added to the queue, and according to their distances to the destination, the best one will always be added to the final step list. The algorithm is implemented by the combination of *start* and *move* methods inside the *Greedy* class.

2.2 Avoiding No-Fly Zones And the Central Area

The ways to avoid no-fly zones and prevent the re-entering to the central area while going to the restaurant are implemented by methods *addNeighbourNode and canAddNodeToOpen* in the *Greedy* class. The movement of the drone is represented by an inner object *Node* defined in the *Greedy* class. Node contains 6 attributes, and they are their positions *lnglat*, their parent nodes *parent*, their distances to the destination *H*, the boolean ***isInCentral***, the directions they are facing to from their parents *direction* and ticks *ticksSinceStartOfCalculation*. If the current node is out of the central area, those possibilities for next movements which enter the central area will be dropped out. Also we have to check the condition that the next movement and the current position form a line that intersects with the central area at the corner, but the next node is not in the central area. We only need to check whether the line has an intersection with one of the edges of the area. **Note that we only need to find the path from the start to the destination and the drone just follows the same route when going back so that we do not have to consider the case for returning.**

The algorithm for avoiding no-fly zones is simple. I just iterate all the 16 possibilities of the next movement and find the valid one which will not touch the no-fly zones and is the closest to the destination, shown as Figure 1.

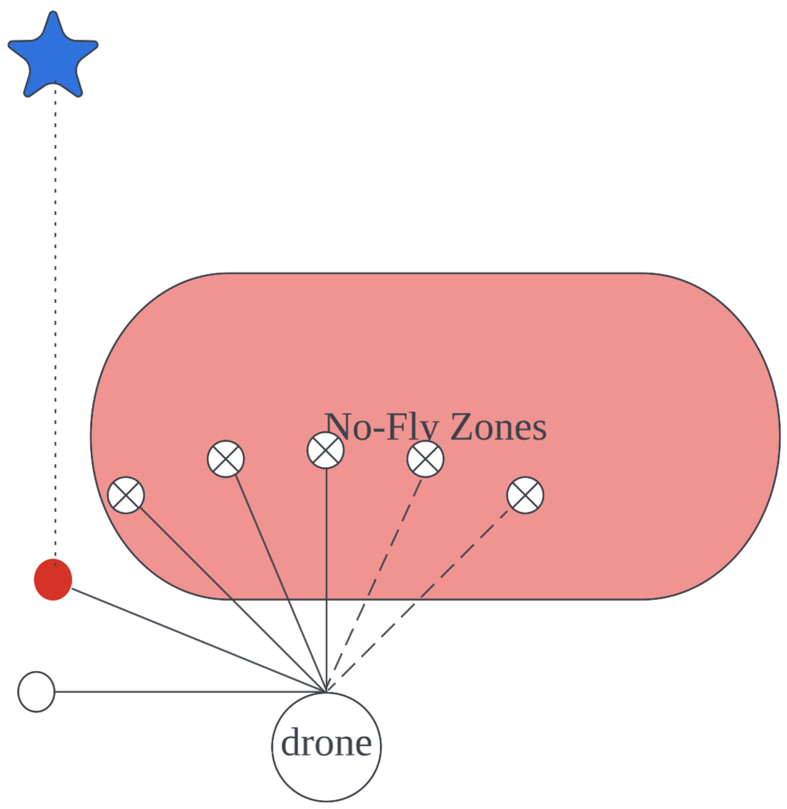

Figure 1: The drone will find the best way that is the closest to the destination without touching the no-fly zones in the meantime.

The checking method for finding the node is called *canAddNodeToOpen*. The algorithm is written as following.

Algorithm 2 canAddNodeToOpen(LngLat lnglat, Node node)

```
1: if !current.isInCentral and lnglat.inCentralArea() then return false
2: end if
3: if current.lnglat.intersectWithNoFlyZone(lnglat) then return false
4: end if
5: return !isLngLatInClosed(lnglat) ▷ Check if the possible movement has been searched and added
   to the path
```

This algorithm basically filters out the nodes which are in the no-fly zones and other invalid conditions.

The method for checking whether the route will go through the no-fly zones is called *intersectWithNoFlyZone* in the LngLat class. This method tries to find whether the line formed by the current node and the next node is going to cross the no-fly zones by finding the intersections of every edge of the forbidden zones. If the intersection exists, the method returns true.

2.3 Delivering Orders As Many As Possible

As the drone has battery, orders in a day cannot all be delivered. Therefore, we need to find a way that the drone can deliver orders as many as possible and the number of successful deliveries is around a sample average. The *sortOrders* method inside the **Order** class is created for doing this job. It sorts all valid orders by the distance from the start to the destination in the increasing trend so that those orders with a close range can have a higher priority for delivering. Moreover, the greedy algorithm can mostly guarantee the path is as short as possible in normal cases, which can help save energy and deliver more orders.

2.4 Routes For Returning

Because the Greedy algorithm can mostly guarantee the path is as short as possible in normal cases, thus the drone solely needs to follow the same route in an opposite direction. The method *finalPath* in the **Drone** class creates an inverse version of the route from the start and combines the path for coming and returning together as a flight path of an order.

2.5 Results

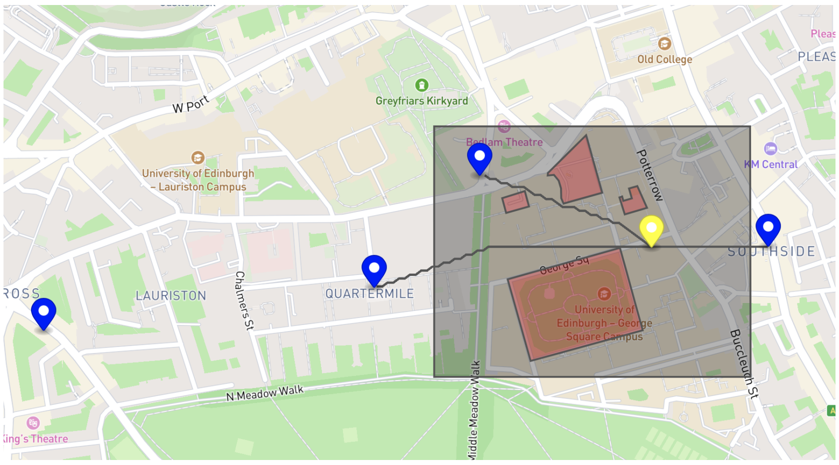

Figure 2: The delivering flight paths for the drone on 2023-04-01

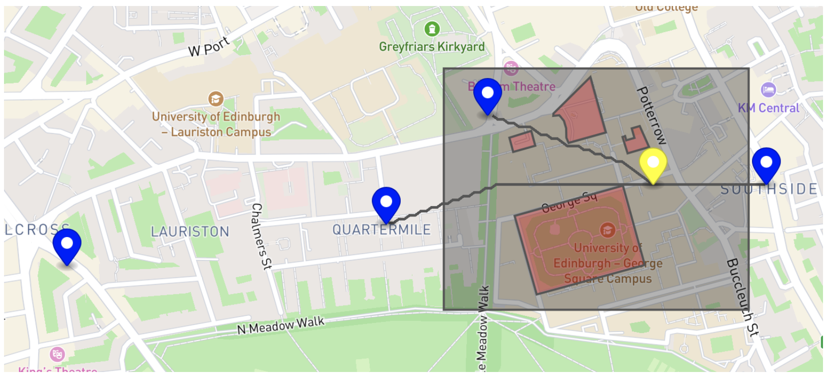

Figure 3: The delivering flight paths for the drone on 2023-02-03

From the figures we can find the farthest restaurant can never be reached by the drone because of its battery restriction.

3 Dependencies and Bibliography

- Luhn Algorithm, https://www.geeksforgeeks.org/luhn-algorithm/, Accessed on 2022-11-29.
- Greedy Algorithms, https://www.geeksforgeeks.org/greedy-algorithms/, Accessed on 2022-11-29