Informatics Large Practical Report

Gleb Sokolovski
S2015488
25 November 2022

Contents

1 Introduction

This document is the description for a pizza-delivering drone. The drone can only move in 16 main compass directions and hovering with a magnitude of 0.00015 per unit of movement. The drone has 2000 moves before it runs out of battery. Its starting and ending point must be Appleton Tower with hardcoded coordinates (-3.186974, 55.944494). Given a particular date, the application communicates with a RESTful API to pull information on orders, restricted flight zones, restaurants and their menus, and "central area" coordinates. It calculates the path that the drone should take before it runs out of charge; and outputs three files that describe the outcome of each order, and the coordinates of its flight trajectory in both JSON and GeoJSON formats.

This document is split into two sections: Software Architecture Description and the Drone Control Algorithm.

2 Software Architecture Description

My solution has eight Java classes, two Enum classes, and a Record - each one has a specific function. In this section, I describe the functionality of each class along with key functions that are necessary for the making of the final design.

2.1 Main Class: App.java

<u>Objective</u>: Initialise the drone class, produce the desires output files.

App.java is responsible for extracting the URL and date provided by the command line, initialising the Drone class and passing on the information provided through the command line, and after all the calculations are done - creating the deliveries-*YYYY-MM-DD*.json, flightpath-*YYYY-MM-DD*.json, and drone-*YYYY-MM-DD*.geojson files.

2.2 Database

<u>Objective</u>: Communicate with the server and retrieve necessary information
for our calculations.

This is a class that's main function is communicating with the REST server.
The initialiser is synchronised for thread safety. This is needed to not destroy the singleton pattern when accessed by multiple threads simultaneously as the different threads will get different instances of the singleton class, therefore defeating the whole purpose of having a singleton class to begin with. There are many ways to ensure thread safety but "synchronized"is the easiest one [2] and it meets our demands so that's the one I chose.

Contains functions to get:

- Restaurant and Menu from '/restaurants'
- Central Area coordinates from '/centralArea'

- Orders from '/orders'
- No fly zones from 'noFlyZones'

Each function uses Gson package to read the JSON from RESTful API and decentralise it into specific classes using the fromJson() function.

2.3 CompassDirection and LngLat

Objective: Initialise points in 2D space; limit direction of movement of drone.

CompassDirection is an Enum class with 16 main compass directions as specifies in the manual and "Hover" - for when the drone is hovering to either drop off or pick up an order. LngLat is a Record class, which works closely with CompassDirection. Its main responsibility is to initialise a point given a longitude and latitude both as double (lng acting like an x-coordinate and lat - as y-coordinate).
Important functions that are used throughout the application include:

- inCentralArea(): checks if a point is inside Central Area - after fetching the Central Area coordinates from the server.
- distanceTo(): checks distance from the current LngLat to a given one.
- closeTo(): checks if current point is within the 0.00015 range from a given point.
- nextPosition(): returns the coordinates of the next position of the drone from current position given a CompassDirection.

all functions are described in more detail in JavaDoc

2.4 Menu and Restaurant

Objective: Provide structure to restaurants and menus pulled from server.

Each restaurant has a name, coordinates and an array of Menu.
Menu consists of just pizza name and price in pence.

Important functions that are used throughout the application include:

- getRestaurantFromRestServer(): returns array of Restaurant fetched from the server.

2.5 OrderOutcome and Order

Objective: Check validity of orders.

OrderOutcome is an Enum class, the same to the one provided to us in the manual.

The Order class defines the structure of each order pulled from the server.
Important functions that are used throughout the application include:

- orderChecks(): checks the validity of orders and updates OrderOutcome accordingly. The list of checks and outcomes are as follows:

Checks done on the order	**Outcome when the conditions are not met for the check**
Checks expiry date of the credit card is after date of order. Since the expiry date is given in the format: MM/yy (i.e. no day), the function takes the last day of the month as the actual date (e.g. 12/2024 —> 31/12/2024) using the Calendar class. The class makes sure to take into account leap years	InvalidExpiryDate
Checks CVV - the length is either 3 or 4 and each digit is a number between 0 and 9	InvalidCvv
Checks total stated matches the delivery cost by using getDeliveryCost() from first part of coursework	InvalidTotal
Number of pizzas ordered. Has to be at most 4 and at least 1.	InvalidPizzaCount
Pizzas ordered have a valid combination of restaurants	InvalidPizzaCombinationMultipleSuppliers
Credit card number given is valid. Checks that length is exactly 16 digits long and then performs the Luhn algorithm [1].	InvalidCardNumber

If all the above checks are passed, the order updates its OrderOutcome to ValidButNotDelivered.

All the dates use the SimpleDateFormat class to get formatted and parsed into a Date.

2.6 NoFlyZone

Objective: Provide structure to restricted flight zones pulled from server.

Consists of name and list of LngLat coordinates that make up the polygons of restricted areas.

2.7 Flight

Objective: Compute the flight trajectory given a start and an end point.

The Flight class has a starting point and an ending point that are given. It calculates the path of the drone and updates the:

- path (List<LngLat>),
- directions (List<CompassDirection>), and
- movesTaken (the moves this particular Flight takes to get from starting point to ending point).

The class takes into account no fly zones and central area, which is described later in the Drone Control Algorithm section.

Important functions that are used throughout the application include:

- calculatePath(): computes the path in as minimal steps as possible while from start to end points; adds a hover at the end to drop off or pick up order.

All other functions are private and are only used inside the class.

2.8 Drone

Objective: Control the path of the drone on any given day.

The class goes through each order for that day and checks that they are valid. The invalid orders are added to the cancelledOrders list. While the valid ones then proceed to have their flight calculated - first from Appleton Tower coordinates to the restaurant and then back - and that's counted as one flight, as we need to know whether we can return home without having battery run out (that is why the array list that stores all the flights is of form: List<List<Flight>>).

Drone class is initialised through App.java, given a URL and a date. From there we can see which orders were cancelled or what's the path of the drone for the day with simple commands.

3 Drone Control Algorithm

Usual path finding algorithms include Dijkstra, A*, Heuristic search, Depth-First, Breadth-First, among others. These would be good algorithms, however since we are restricted in movement by 16 directions, these algorithms could take as long as $O(16^n)$, with space complexity really bad as well.
Since speed is key in a drone delivery system, I've decided to go for a different approach, which I describe below.
The control of the flight is done mainly by the Flight and Drone classes.

3.1 Avoiding No-Fly-Zones

3.1.1 Problem Identification

As mentioned previously, the Flight class is responsible for computing the path between any 2 given points: starting point (SP) and ending point (EP).

First thing we do is calculate the direct line (i.e. the quickest and shortest way) from SP to EP. Then I've split that path into 2 possibilities:

1. There are no no fly zones on the way: we take the straight line path calculated earlier.
2. There are fly zones that need to be avoided: two paths emerge - as demonstrated in Figure 1.

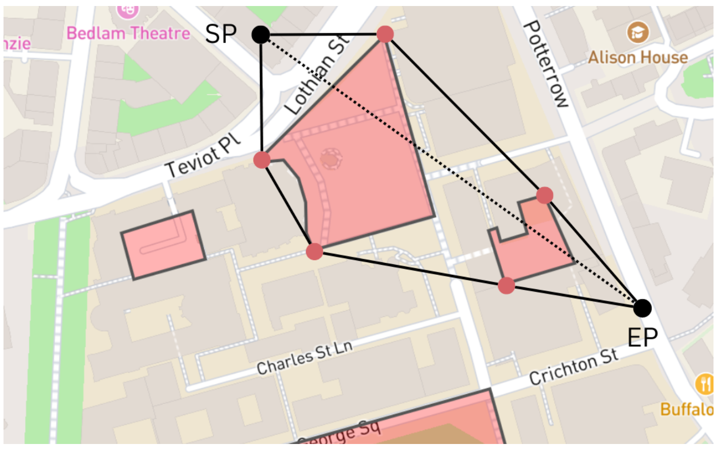

Figure 1 - Scenario 2: Convex-Hull problem

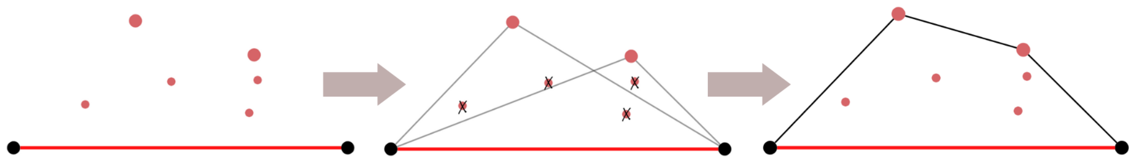

Figure 2 - Visualisation of Simplified Convex-Hull

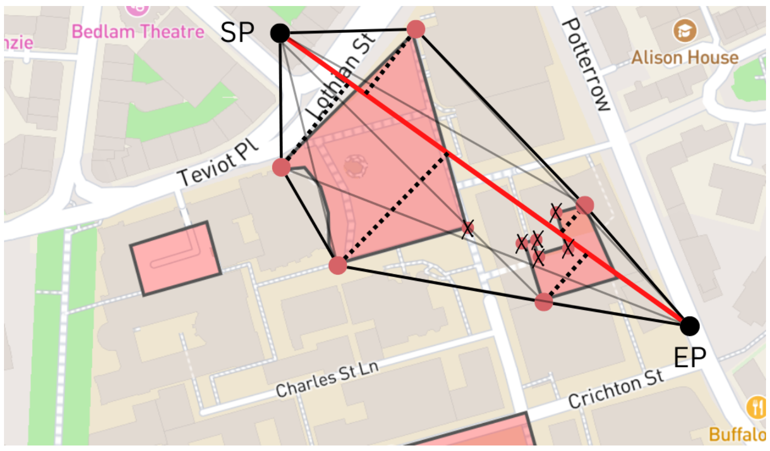

Figure 3 - Convex-Hull Applied to Our Situation

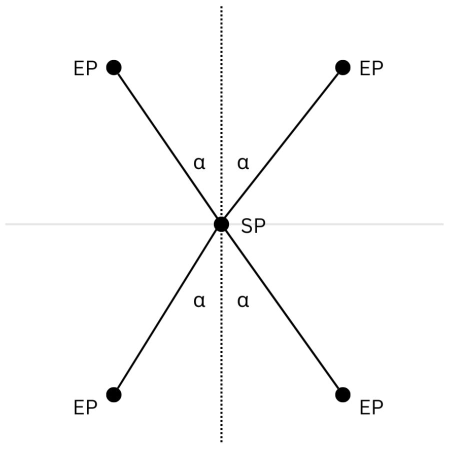

Figure 4 - Angle α for Straight Line Flying

3.1.2 Computational Approach to Solve the Convex-Hull Problem

In our case the list of points (LP) is the coordinates of polygons that make up no fly zones along with SP and EP. The solution to our problem is a divide and conquer algorithm as follows:

1. Draw a line between SP and EP - from now on referred to as the base line.
2. Pick the top and bottom points from LP.
3. Draw two triangles with them and the base line.
4. Remove all the points from inside the triangle.
5. Repeat steps 2-4 until LP is empty.
6. The points left are the points that outline LP.

A simplified visual representation is shown in Figure 2.

Pseudocode Solution:
Input: List of no-fly-zone coordinates LP, start point SP, end point EP
Output: List of <List of Top Outlines and List of Bottom Outlines>

```
baseLine = lineBetween(SP, EP)
outlines = []
topRoute = []
bottomRoute = []
while LP ≠ Ø:
      topPoint = topByLat(LP)       // highest in lat
      bottomPoint = bottomByLat(LP) // lowest in lat

      add topPoint to topRoute
      add bottomPoint to bottomRoute.add

      topTriangle = baseLine + topPoint
      bottomTriangle = baseLine + bottomPoint
      for each point in LP:
            if point inside topTriangle OR
                                point inside bottomTriangle:
                  remove point from LP
add topRoute to outlines
add bottomRoute to outlines
```

3.1.3 Finding a Straight Line Path Between two Points

Now that we have all the points we need for the quickest route, now we need to define the algorithm for drone control to fly the straight line between two any given points. We call the angle between the two points α. We define it as shown in Figure 4.

If α coincides with one of the main compass directions, then it's simply just travelling at that compass direction until we are close to the end point. However let's imagine the scenario from Figure 5.

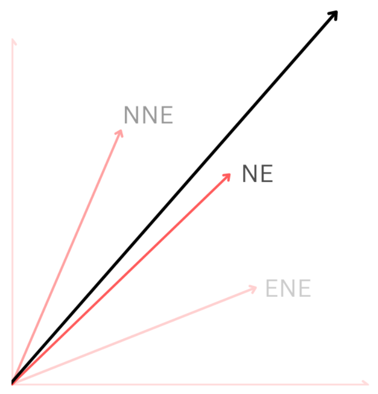

Figure 5 - Angle ɑ Doesn't Coincide with One of the Main Compass Directions

Our fastest route is between two possible directions. In this situation we create two variables: *bestDirection* and *secondBestDirection*. In the example from Figure 5, *bestDirection* = NE and *secondBestDirection* = NNE.
The reason I keep track of second best direction to take is in case the best direction doesn't meet the requirements. There are three reasons why our best direction would be considered "wrong":

1. It leads to a no fly zone,
2. It leads to outside of Central Area before the order is delivered, or
3. It leads to inside of Central Area before the order has been collected outside of Central Area.

We use LngLat's nextPosition() and Flight's inNoFlyZone() to check for this criteria and if one of them is triggered, we use the second best direction.

It is valid to point out that because we are dealing with such small steps (0.00015 deg), we don't expect this to happen often. Each inNoFlyZone()
takes an $\mathcal{O}(n \cdot m)$ time complexity (where n is the number of no fly zones and m is the number of coordinates for the no fly zone). Since we call it for each step, we will have the time complexity of $\mathcal{O}(n \cdot m \cdot p)$, where p is the number of steps for the flight.

Even though n and m are not large, p is so it takes a significant chunk to do the calculation. Since it's vital for successful completion, I've kept it in the source code, however if I had more time, I'd improve it by adding a function which checks whether the next step is anywhere near a no fly zone or central area boundary and therefore if there is any point in checking whether we have crossed into it. Therefore decreasing the the time complexity of that check down to $\Theta(n \cdot p)$.

3.2 Choosing the Order of Deliveries to Maximise Number of Deliveries Made per Day

As mentioned earlier, the Drone class is responsible for putting everything together. It calculates the flight path for each valid order. It then looks at how many steps those orders will take.
If it's below 2000, no further calculation is needed and we proceed to deliver them all and changing their OrderOutcome to "Delivered".

On the other hand if the step count is above 2000, we need to figure out what the most efficient way of delivering pizzas is. There are two functions in the Drone that achieve this:

1. finaliseFlightsForTheDay(): Rearrange the Orders in ascending order of steps. Then start delivering them as long as they keep the step count below 2000. If the next order will push it over the limit, we move it and the rest of orders to the *cancelledOrders* list.
2. notOrderedFinalise(): Exactly the same but don't arrange the Orders in ascending order of steps - leave them in the same order we pulled them from the server.

The results are displayed in Figure 6:

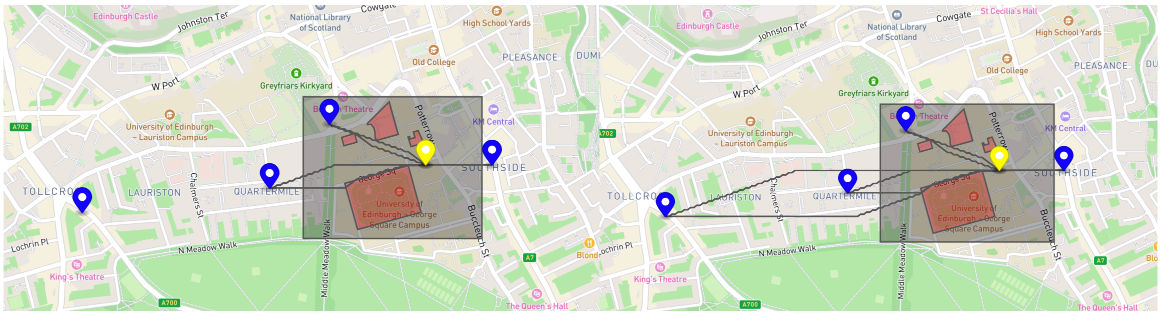

Figure 6 (a) - 2023-03-30

Left is finaliseFlightsForTheDay(): Orders delivered: 30 out of 41 in 1922 moves

Right is notOrderedFinalise(): Orders delivered: 20 out of 41 in 1982 moves

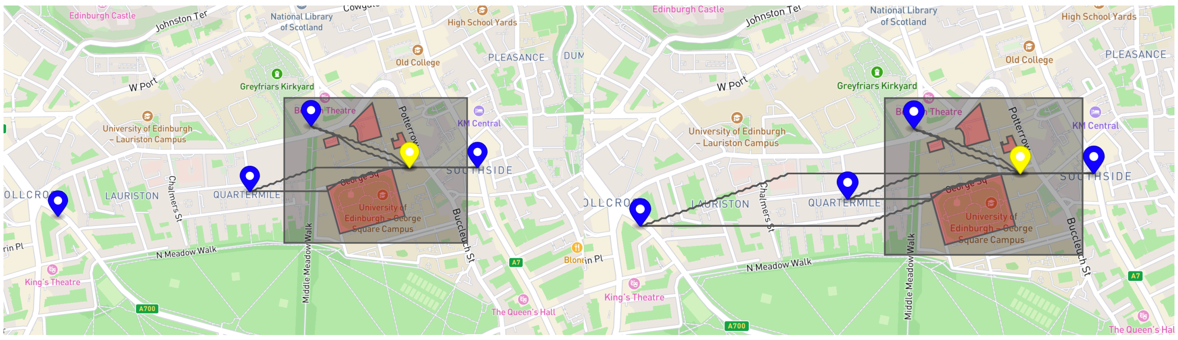

Figure 6 (b) - 2023-01-01

Left is finaliseFlightsForTheDay(): Orders delivered: 31 out of 42 in 1984 moves

Right is notOrderedFinalise(): Orders delivered: 19 out of 42 in 1788 moves

4 Conclusion

The application runs on average around 29 seconds.

I have already mentioned a possible improvement point for detection of near by no-fly-zones for quicker calculation times in Section 3.1.3.

Out of the two functions mentioned in Section 3.2, finaliseFlightsForTheDay() is definitely better but obviously can't be used once we have introduced time of order - in which case whoever ordered first deserves to have their order delivered first.

Since we are dealing with a limited number of of restaurants, a great way to improve the current algorithm could be to calculate path to each restaurant and then manipulate it from there. At that point, it would be sensible to consider a cost-reward analysis and realise at what number of restaurants does this become too computationally expensive and time wasting - for example if we have every restaurant in Edinburgh, while pizzas are ordered only from three of them. Since I don't know how the application is going to be tested and ran, I've decided to hold off that method.

References

[1] Luhn Algorithm: https://en.wikipedia.org/wiki/Luhn_algorithm
[2] Thread Safety: https://www.digitalocean.com/community/tutorials/thread-safety-in-java