Informatics Large Practical

Description of the Designed Application

s1911792

December 4, 2021

Contents

1 Software Design Description

This section is to provide a brief description on the architecture of the software.

1.1 Implementation Task

The implementation is aimed to set up an ordering system that can deliver food from different shops to the desired locations, while bypassing no-fly zones, before the deliver drone runs out of battery. It should be back to where it started if it delivered everything, or about to run out of the battery.

1.2 Description on the Application

My implementation consists of 14 classes:

1. `App.java` The class that would run the application.
2. `AllOrder.java` The class that would combine the paths for each order together, and gives the path and total money earned on the day.
3. `DerbyIO.java` The class that handles Derby database requests, such as reading and writing to the database.
4. `GeoJson.java` The class that controls the read and write from/to a GeoJson file.
5. `Heuristic.java` The class that implements an admissible heuristic for a* algorithm
6. `Location.java` It converts a "what3word" string into a `LongLat` object.
7. `LongLat.java` This class stores the location of each object, as well as the angle of the drone at each location. It also contains various method that would do coordinate related arithmetics.
8. `Menus.java` The `Menus.java` would handle the prices of each order.
9. `NoFlyZone.java` This class is responsible for handling reading and calculating no-fly zones, as well as calculating points that's close to the no-fly zones.

10. `Order.java` The `Order` class would read all the orders' delivery points as well as shops and store them in a given format.

11. `Request.java` This class handles all the `HttpRequest` set-ups.

12. `SingleOrder.java` This class would compute each single order's delivery path by using a*, as well as converting the path into moves.

13. `Details.java` and `Shop.java` This two classes are to set up type tokens, for us to read values from server.

1.3 UML Diagram of the structure and Basic Interactions

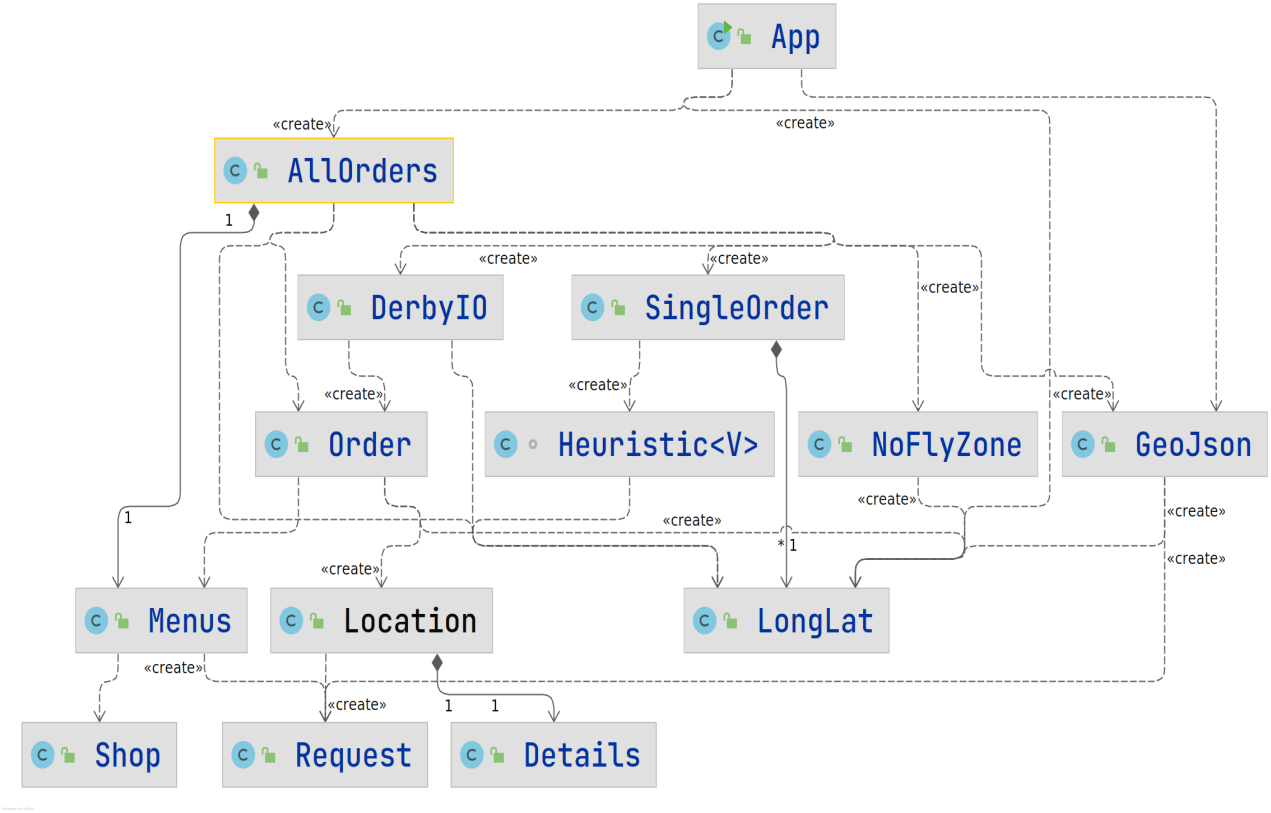

Figure 1.1: UML Diagram of the design

The application would take user-input date (`App`), read the required data from the web server and database(`DerbyIO, GeoJson, Order, Location`)), setting up a graph(`NoFlyZone, SingleOrder`) and compute the route for the drone(`AllOrders,SingleOrder,Heuristic`). Then the drone would simulate the optimal path given by a* algorithm move by move.The output will be a geojson file that contains all the moves, and user messages describing the performance of the delivery.

2 Drone Control Algorithm

The idea of drone control algorithm is quite simple. For each order we calculate the path from given start to given end.

2.1 Graph and Searching

The graph's vertex consists of three parts: starting point, way points, shop(s) and ending point.To generate way points, I used the points that's close to the no-fly zone corners. For the edges on the graph I used Manhattan distance to set them up, if and only if they don't cross the no-fly zone.

2.1.1 Close Points

For each corner we generate 4 points in quadrant (the current corner will be considered as $(0, 0)$), using an appropriate angle, with a scaling parameter. In my case I choose the angle to be $\pi/6$ and scaling parameter to be 2.75, so that the points will be separate and far enough for drone to adjust its path, when moving on the path given by a*.

2.1.2 A* Path Finding

The algorithm that I used for path-finding problem with obstacles, in this case no-fly zones, is A*. A* algorithm is famous for doing quick path-findings. It is an informed search algorithm on weighted graphs: given a starting point, it tries to find a traversing path to a destination with the least cost. It is done by extending paths on a tree of paths rooted at the starting vertex, until destination is reached.[1] In my implementation I used JGraphT's[2] package for doing a* path finding.

To make sure that we could have an optimal solution I implemented an admissible heuristic using Manhattan distance, so that it will never overestimate the cost.

The implementation of computing all possible paths are shown below:

Algorithm 1 Drone Control Algorithm For a Single Order

```
1: for order in Orders do
2:     for vertex in Graph do
3:         a*(start, end, path)
4:     if shop.size() == 2 then
5:         var1 ←path(start, shop[1], shop[2], end)
6:         var1 ←path(start, shop[2], shop[1], end)
7:         if var1 ≤ var2 then
8:             use path(start, shop[1], shop[2], end)
9:         else
10:            use path(start, shop[2], shop[1], end)
11:    else
12:        use path(start, shop[1], end)
       add allPaths ⟵ path
13: concatenate(allPaths)
```

2.1.3 Drone Movement

After all the paths are calculated, the drone would try to move on the paths given by a*. It will trying to turn to the vertices on the path and move to that position. After finishing the current path, starting point will be set to the current ending point.

Below shows a path that goes from previous delivery point to the next delivery point:

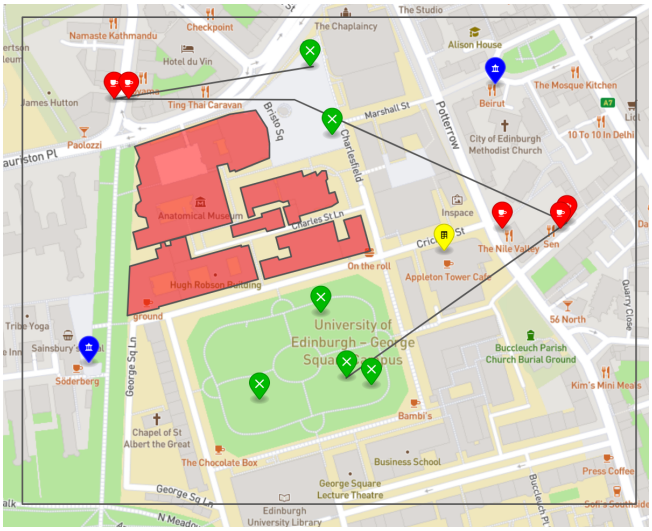

Figure 2.1: A delivery path generated by a* as shown

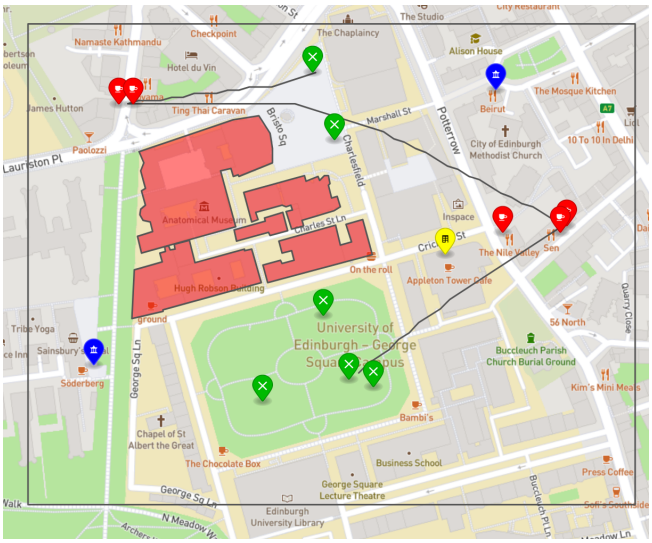

Figure 2.2: A delivery path simulated by drone as shown

2.2 The Order of Orders

By the specification, we are restricted to deliver one order at a time. Hence a major question arise:in which order should I deliver the orders? To solve this problem, I did a simple greedy approach, by going to the closes shop with smallest Manhattan distance to current starting point. Using Manhattan distance instead of Euclidean distance is to approximately represent the cost of crossing to no-fly zone.

I have not calculate all the possible path size for the upcoming order,

inasmuch as that it will be computationally expensive to do so, even for a* path finding.

2.3 Instability of the Algorithm

The application will produce different results. This may be caused by admissible heuristic is not consistent, i.e. in some cases it will overestimate the cost.

To fix this problem, I let it run for 15 times and get the best outcomes of the run. This can be done because a* path finding is fast enough for me to do so. And this is another reason that I used greedy approach to decide the order of the orders, i.e. to reduce the computational complexity.

In such way,it is possible to having a very optimal output on the hardest day (2023-12-27), in the given data set, as shown below:

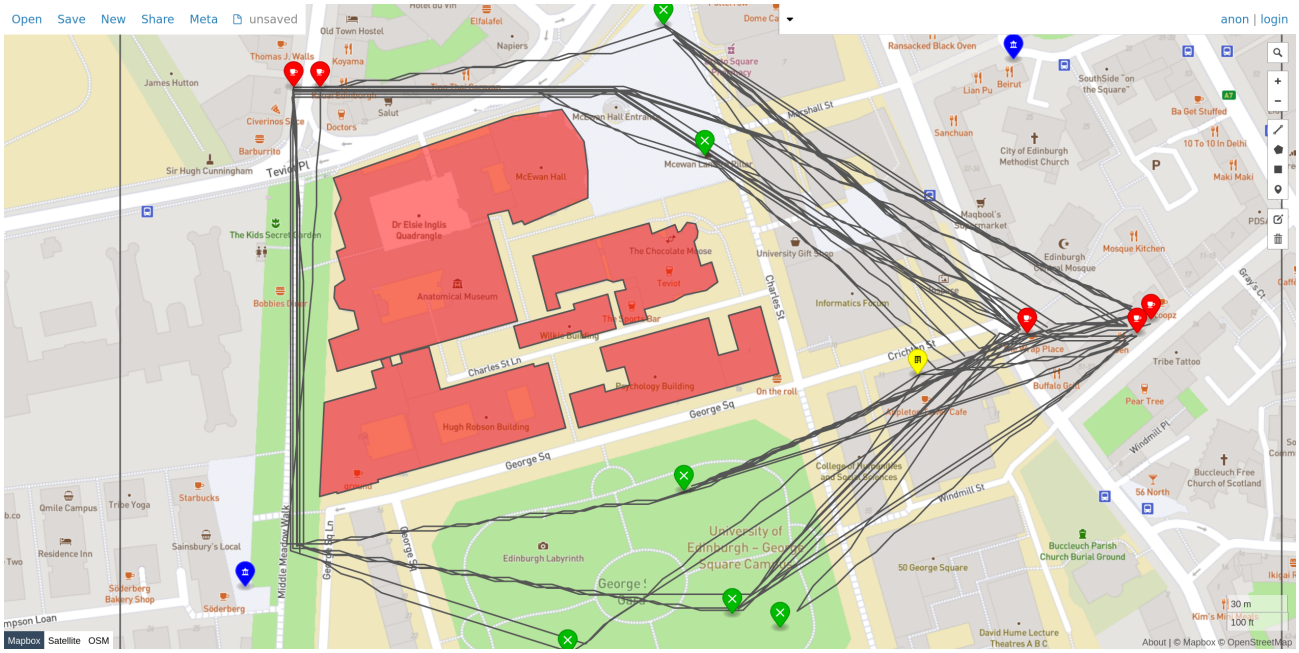

Figure 2.3: Delivery path for 2023-12-27

This is with an monetary value of 92% delivered. Note that it's not possible to deliver all of the orders in this sample case.

3 Advantages and Disadvantages

The advantages of this implementation is that it's very optimal comparing to most of the other implementation. It goes to points that's very close to the no fly zone, and a* path finding makes sure that it would go to the optimal paths.

However, the space complexity isn't something that can be ignored. While a* algorithm already having $O(\mid V \mid)$ space complexity, by running it multiple times might not be possible on a ram-limited server.

Another major downside is that it might hit the no-fly zone. In contrast to the previous one, this problem can be solved by adjusting the distance between close points to the no-fly zone corners and corners themselves. Adding the distance would solve this problem.

A side note is that this algorithm does not need to check the confinement due to the close points are unlikely to be in the confinement area, practically. If there happened to be an issue with the confinement area, i.e. there exists an strictly confined area for the drone to operate, I would suggest adding confinement area as no-fly zone in order to properly operates.

References

[1] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. “A Formal Basis for the Heuristic Determination of Minimum Cost Paths.” In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. DOI: 10.1109/TSSC.1968.300136.

[2] JgraphT. *JgraphT: a Java library of graph theory data structures and algorithms.* URL: https://jgrapht.org/. (accessed: 27.11.2021).