Informatics Large Practical

Coursework 2 Report

S1923846

Andy Fu

1 Software Architecture Description

1.1 Java Classes

There are 12 classes in total.

- *App*: This is main class of the project; it contains the main method with takes command line arguments as input and a method for writing *Geojson* file as output.
- *Const*: The class has all constants for the project; thus, any modification of constants can be done easier.
- *DatabaseConnection*: The class handles connection with the database server, it can read data from or write data into the database.
- *Drone*: The class controls the drone; it picks which order to deliver next and generate a flight path. If the system upgrades and we have more drones to control, we can simply create more drone objects.
- *HttpConnection*: This class handles connection with the Http server, it can also parse *Json* and *Geojson* files. As the database and those files are in two sperate servers, so it would be neater to have two classes for each server.
- *Item*: Class required for parsing Json files, stores details of a food item.
- *LongLat*: The class representing a point on the map, and related computations like distances from another points. As there are many computations related to coordinates, they need an own class, otherwise classes using these computations can be messy.
- *Map*: The class representing the map which holds all geometrical information. Path finding algorithm is carried out in this class, as the algorithm is complex, writing it in other classes will significantly reduce the readability.
- *Node*: This is required for A* search algorithm in Map class, it represents a linked list of states, with cost, heuristic and angle of last move.
- *Order*: This class stores details of an order, computation of utility (for choosing deliver which order first) is carried out in this class.
- *Path*: This class stores details of a move, which will be stored in the database.
- *Shop*: The class is required for parsing Json files, stores details of a shop.

There is one main class, one classes for storing all constants, two classes for connecting two servers. Drone, LongLat (coordinates), Map, Order are real-world objects, they have functions that are related to themselves. Node is required for A* searching algorithm. Shop and Item are required for parsing json files. Path is used for store database output. I think all classes are necessary, and none of them should be split into multiple classes, as they will likely to be abstract.

The path finding algorithm is mainly processed in Map, due to the complexity of the algorithm, Map class may have lower readability than other classes.

1.2 UML Class Model

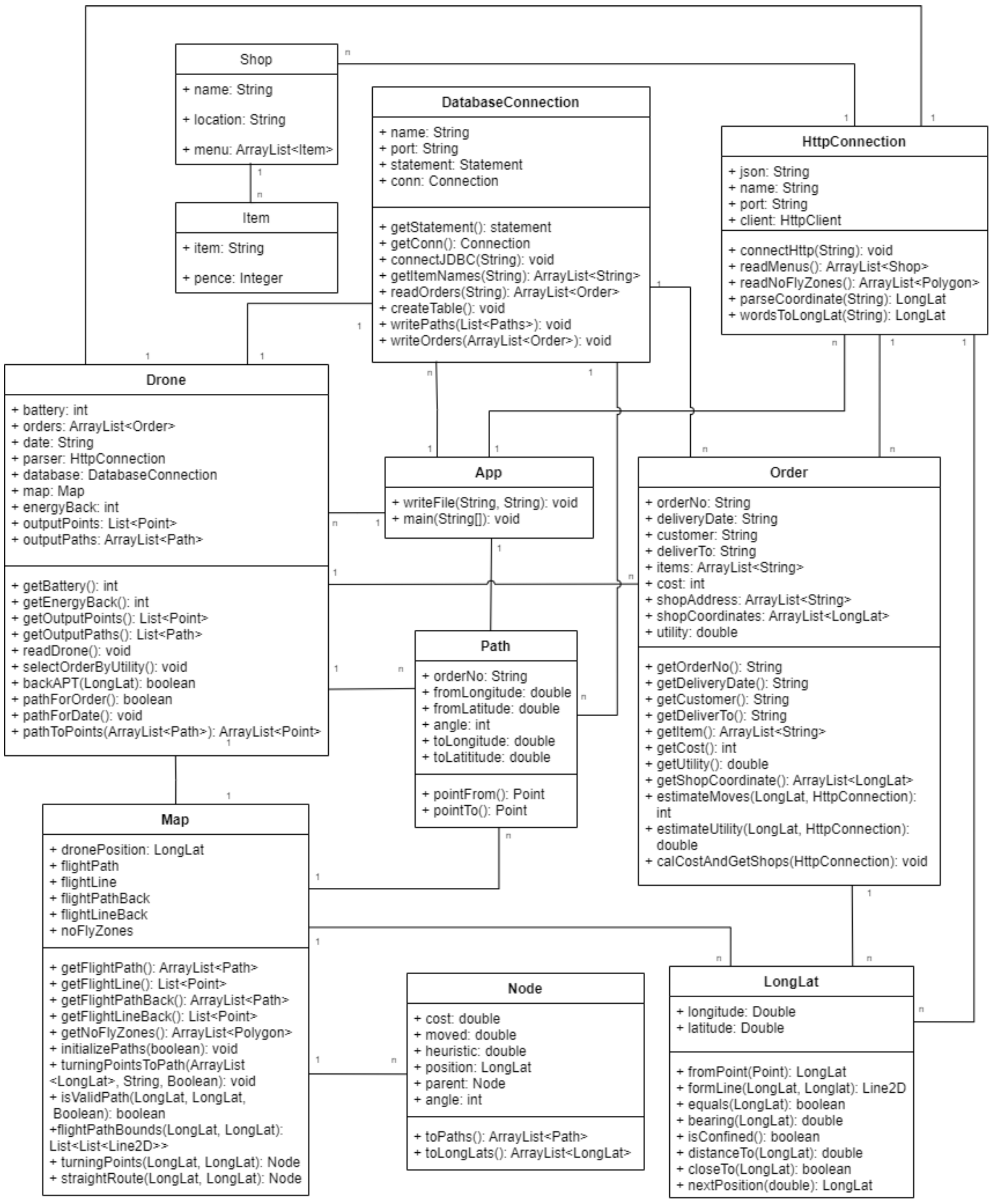
Shop
+ name: String
+ location: String
+ menu: ArrayList<Item>
Item
+ item: String
+ pence: Integer
DatabaseConnection
+ name: String
+ port: String
+ statement: Statement
+ conn: Connection
+ getStatement(): statement
+ getConn(): Connection
+ connectJDBC(String): void
+ getItemNames(String): ArrayList<String>
+ readOrders(String): ArrayList<Order>
+ createTable(): void
+ writePaths(List<Paths>): void
+ writeOrders(ArrayList<Order>): void
HttpConnection
+ json: String
+ name: String
+ port: String
+ client: HttpClient
+ connectHttp(String): void
+ readMenus(): ArrayList<Shop>
+ readNoFlyZones(): ArrayList<Polygon>
+ parseCoordinate(String): LongLat
+ wordsToLongLat(String): LongLat
Drone
+ battery: int
+ orders: ArrayList<Order>
+ date: String
+ parser: HttpConnection
+ database: DatabaseConnection
+ map: Map
+ energyBack: int
+ outputPoints: List<Point>
+ outputPaths: ArrayList<Path>
+ getBattery(): int
+ getEnergyBack(): int
+ getOutputPoints(): List<Point>
+ getOutputPaths(): List<Path>
+ readDrone(): void
+ selectOrderByUtility(): void
+ backAPT(LongLat): boolean
+ pathForOrder(): boolean
+ pathForDate(): void
+ pathToPoints(ArrayList<Path>): ArrayList<Point>
App
+ writeFile(String, String): void
+ main(String[]): void
Order
+ orderNo: String
+ deliveryDate: String
+ customer: String
+ deliverTo: String
+ items: ArrayList<String>
+ cost: int
+ shopAddress: ArrayList<String>
+ shopCoordinates: ArrayList<LongLat>
+ utility: double
+ getOrderNo(): String
+ getDeliveryDate(): String
+ getCustomer(): String
+ getDeliverTo(): String
+ getItem(): ArrayList<String>
+ getCost(): int
+ getUtility(): double
+ getShopCoordinate(): ArrayList<LongLat>
+ estimateMoves(LongLat, HttpConnection): int
+ estimateUtility(LongLat, HttpConnection): double
+ calCostAndGetShops(HttpConnection): void
Path
+ orderNo: String
+ fromLongitude: double
+ fromLatitude: double
+ angle: int
+ toLongitude: double
+ toLatititude: double
+ pointFrom(): Point
+ pointTo(): Point
Map
+ dronePosition: LongLat
+ flightPath
+ flightLine
+ flightPathBack
+ flightLineBack
+ noFlyZones
+ getFlightPath(): ArrayList<Path>
+ getFlightLine(): List<Point>
+ getFlightPathBack(): ArrayList<Path>
+ getFlightLineBack(): List<Point>
+ getNoFlyZones(): ArrayList<Polygon>
+ initializePaths(boolean): void
+ turningPointsToPath(ArrayList <LongLat>, String, Boolean): void
+ isValidPath(LongLat, LongLat, Boolean): boolean
+flightPathBounds(LongLat, LongLat): List<List<Line2D>>
+ turningPoints(LongLat, LongLat): Node
+ straightRoute(LongLat, LongLat): Node
Node
+ cost: double
+ moved: double
+ heuristic: double
+ position: LongLat
+ parent: Node
+ angle: int
+ toPaths(): ArrayList<Path>
+ toLongLats(): ArrayList<LongLat>
LongLat
+ longitude: Double
+ latitude: Double
+ fromPoint(Point): LongLat
+ formLine(LongLat, Longlat): Line2D
+ equals(LongLat): boolean
+ bearing(LongLat): double
+ isConfined(): boolean
+ distanceTo(LongLat): double
+ closeTo(LongLat): boolean
+ nextPosition(double): LongLat

Figure 1: UML Class Model

This is the UML class Model of my project, Const class is no included in the class, as it contains only public final constants which can be used by any Class. Showing Const will not provide more information, but can make the graph messy.

1.3 UML Sequence Diagram

This graph shows sequence of how cores classes interact with each other, details and helper functions are not shown.

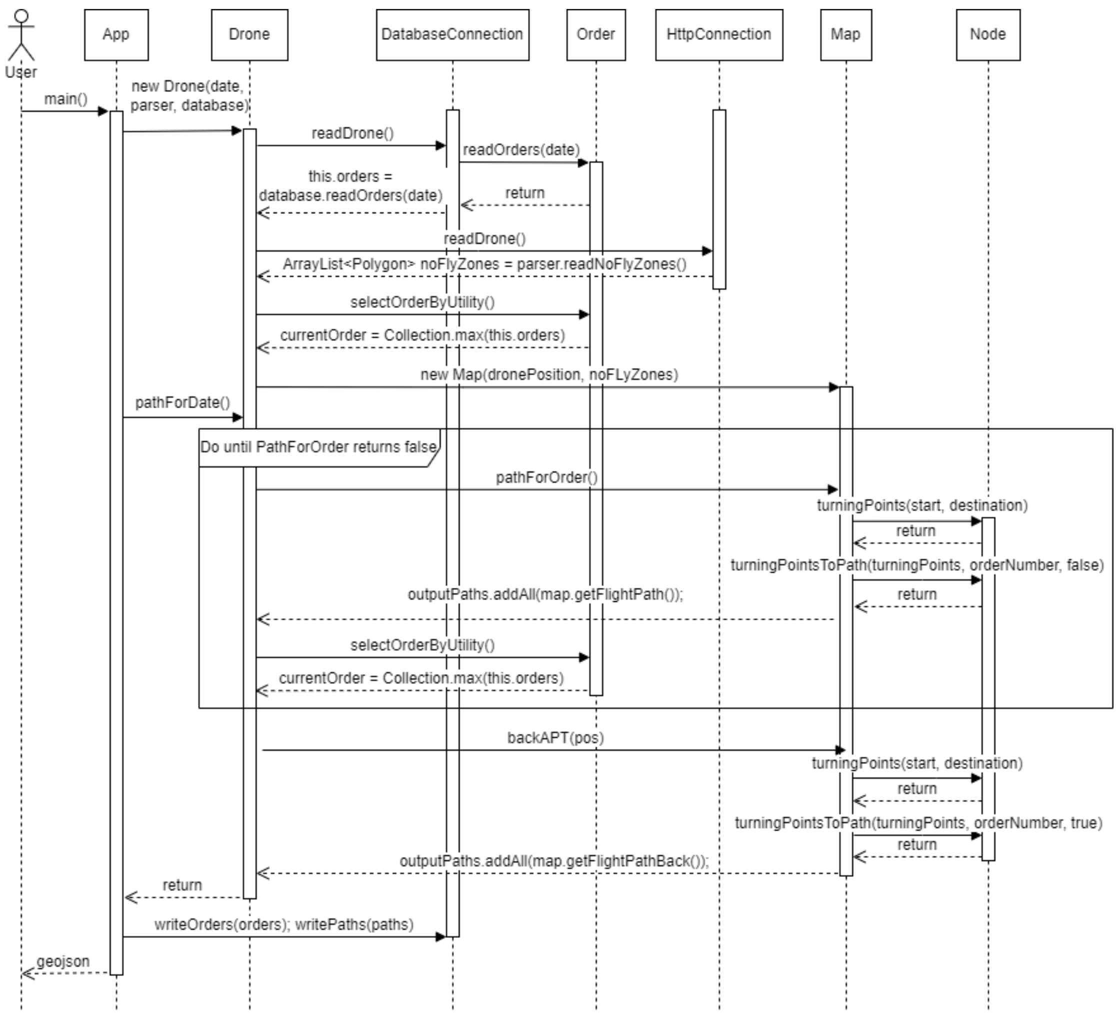
User
App
Drone
DatabaseConnection
Order
HttpConnection
Map
Node
main()
new Drone(date, parser, database)
readDrone()
readOrders(date)
this.orders = database.readOrders(date)
return
readDrone()
ArrayList<Polygon> noFlyZones = parser.readNoFlyZones()
selectOrderByUtility()
currentOrder = Collection.max(this.orders)
new Map(dronePosition, noFLyZones)
pathForDate()
Do until PathForOrder returns false
pathForOrder()
turningPoints(start, destination)
return
turningPointsToPath(turningPoints, orderNumber, false)
outputPaths.addAll(map.getFlightPath());
return
selectOrderByUtility()
currentOrder = Collection.max(this.orders)
backAPT(pos)
turningPoints(start, destination)
return
turningPointsToPath(turningPoints, orderNumber, true)
return
outputPaths.addAll(map.getFlightPathBack());
return
writeOrders(orders); writePaths(paths)
geojson

Figure 2: UML Sequence Diagram

2 Drone Control Algorithm

2.1 Determine Which Order Will Be Delivered Next

The drone prefers deliver expensive order or order which does not use much battery. Some expensive orders may require the drone travels very long, sending those orders can be worse than sending cheaper order which does not require the drone travel long.

Except the first order, the drone always starts delivering each order from the deliverTo location of previous order, so an optimal decision will let the drone deliver to places which near a food shop of the next order. However, if there are n orders, there will be n! possible sequences, finding the optimal solution is a NP problem (even paths are already computed).

As we have limited time, solving the above questions by brute-force is impossible, so I decided to use greedy method. Each order has a utility value, which is the cost of the order divided by the estimated distance for delivering this order. The estimated distance is the sum of straight-line (Euclidean) distance between the drone to shop, shop to shop (only if the order consists of food items in two shops), and shop to customer. If one order has two shops, we compute two distances and select the shorter one (and sort shops simultaneously). The estimated distance is used as we do not have time to compute the path finding algorithms so many times.

As the drone is changing places after each order, thus the distance from the drone to the shop varies, so my algorithm selects a new order only after the previous order is successfully delivered.

2.2 Path Finding Algorithm Introduction

My implementation does not include traveling around landmarks. The path finding algorithm is A* search (but slightly different for reducing time complexity).

To find a path from one point to another in constrained area, we set g(n) as the number of moves to reach a state n, and the heuristic h(n) is the Euclidean distance between current position and the destination, the heuristic is admissible as it never overestimates the true cost, and it is consistent, f(n) can never decrease when searching forwards. As the heuristic is admissible and consistent, the algorithm is guaranteed to produce optimal solution.

A* search is fast when the drone will travel roughly straight (not perfectly straight as it picks angle with multiples of 10 degrees), but it can be extremely slow when the drone needs to bypass no-fly-zones. When the drone is bypassing no-fly-zones, it will search a lot more points, if the drone needs to take a detour with n more moves, the algorithm will search 36^n more points. So, using A* search directly is impossible.

In my algorithm, I assume the drone will only turn at corners of buildings, given the shortest path between two points is a straight line. As shown by Figure 3, the shortest path for a drone (bottom left) to reach customer (top right), only turns at corners of the building.

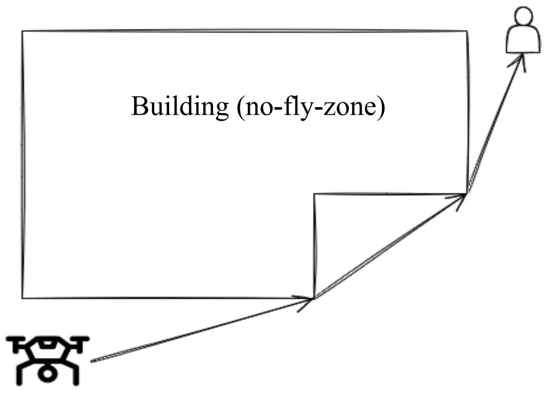

Figure 3: Shortest Path Example

So the first task is to find where the drone turns, and assume the drone will travel straight between two consecutive turning points, when the drone is travelling from one turning point to another, assuming it can travel in any direction.

2.3 Finding Turning Points (The Frame of Flight Path)

Given the drone only changes direction at corners of no-fly-zones, A* search is used in my implementation for finding turning points. List of corners of no-fly-zones are defined in polygons of buildings, the A* search only extends paths that the drone does not fly into no-fly-zones. Heuristic is still euclidean distance, it is admissible and consistent. As a result, the drone is gauranteed to find shortest path if it can fly in any direction and can pass corners of no-fly-zones.

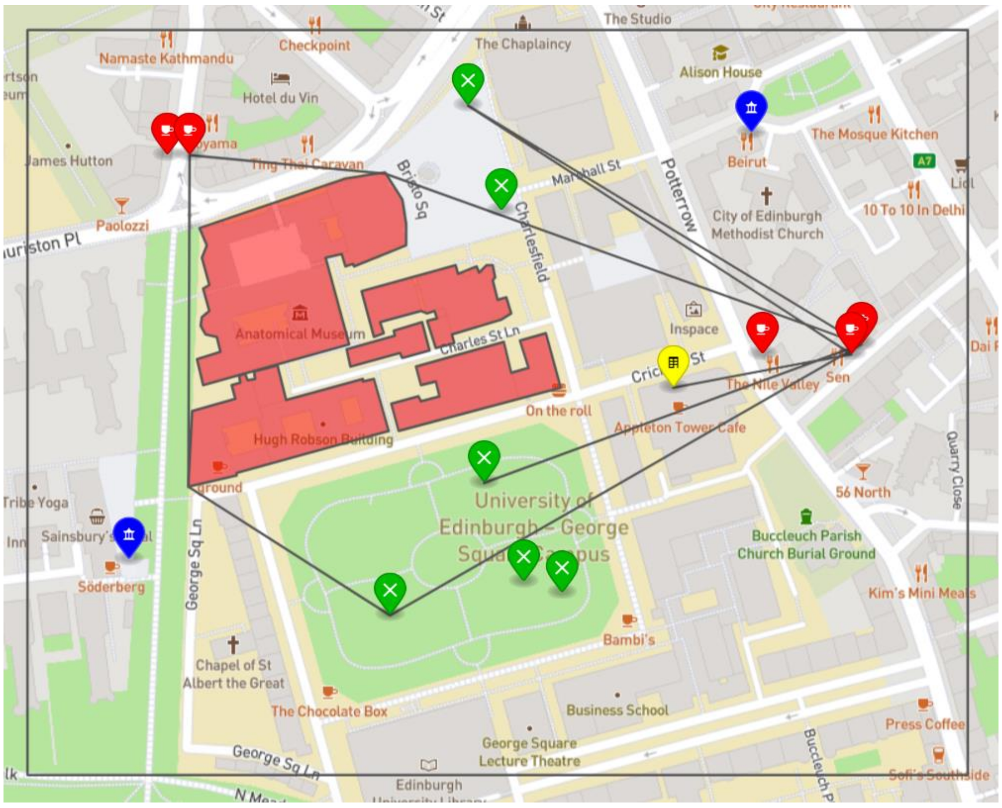

Figure 4: Visualized Turning Points (Frame of the Flight Path)

2.4 Avoiding No-Fly-Zones When Finding Turning Points

Both line-to-line intersection and point-in-polygon is used in this section. This is avoiding no-fly-zones when the drone will fly multiple moves. This refers to function *isValidPath* when parameter singleMove = false.

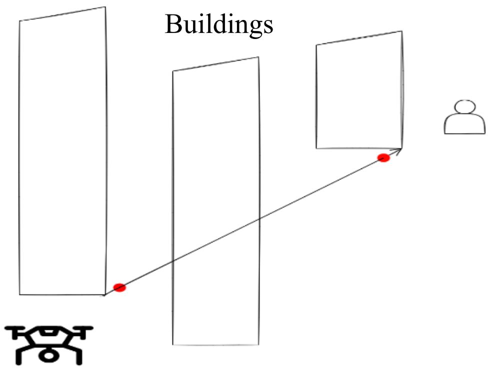

Figure 5: Invalid Path 1

As turning point may have the same position as the corner of no-fly-zones, and that line between these turning points will intersect with lines of no-fly-zone, so I moved two points slightly closer to each other (like the red points in figure 5), then the line does not intersect at vertices. However, line-to-line intersection itself is not enough, Figure 6 shows an invalid path which does not intersect with any lines of no-fly-zones. So, we also need to consider point-in-polygon problem. If one red point in any polygon, the path is invalid.

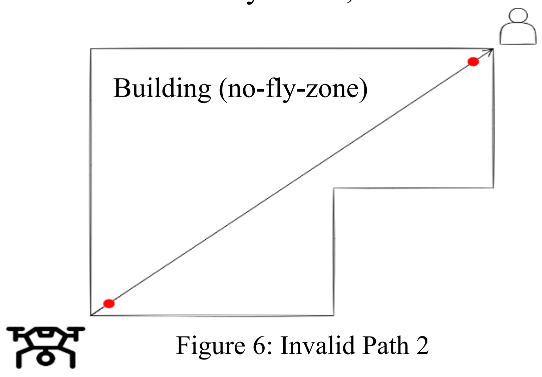

Figure 6: Invalid Path 2

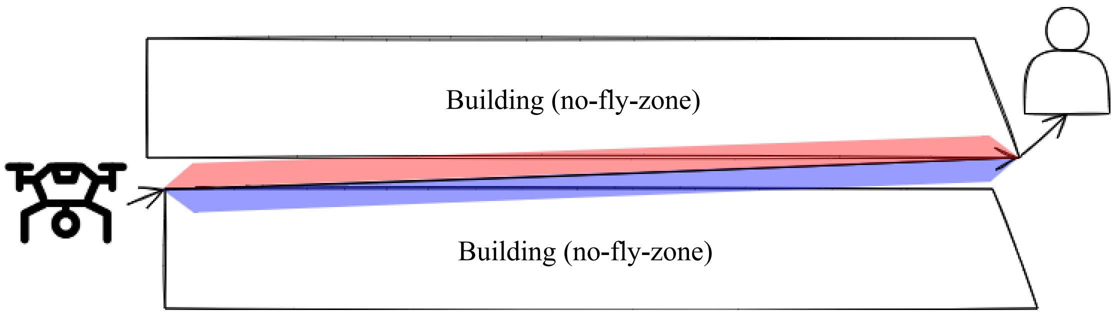

Figure 7: Small Gap Between Two Buildings, and Flight Path Boundaries.

There are can still be exceptions, the flightpath is usually a zigzag line, when the drone is trying to reach a place with bearing angle that is not multiple of 10 degrees. The drone cannot pass very small gaps. As shown by Figure 7, the drone cannot pass the gap between two buildings. To avoiding the drone fly into very small gaps, I implemented a function to estimate the boundary of the flight path, the flight bounds are represented by red and blue trapezoid, the drone can fly only inside each of two trapezoids. The laterals' length is one move of the drone. The angle between lateral side and the longer base is 10 degrees. If both trapezoids (trapezoids are computed in function *flightPathBounds*) intersect (at least one edge of the trapezoid intersects) with any edge of no-fly-zones, the path is passing a small gap, and it should be invalid.

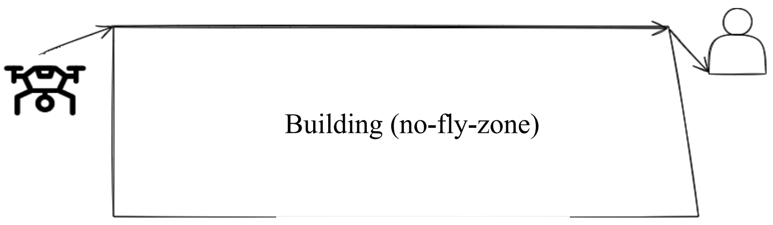

Figure 8. Valid Path

Furthermore, if the path is coincided with any edge of no-fly-zones, the path is valid. As shown in Figure 8. If both vertices of the path on a same edge of no-fly-zones, then the path is coincided with that edge.

The line-to-line intersection problem is solved using library java.awt.geom.Line2D. The point-in-polygon problem is solved using library com.mapbox.turf.TurfJoins.

2.5 Finding Full Path after Found All Turning Points

After finding all turning points, we can apply A* search directly between each two consecutive turning points, as the drone will go roughly straight, so the result can be computed very efficiently. The A* search is the same as described at the beginning of section 2.2, except it only consider half (bearing angle ± 90 degrees) of possible directions, as the drone will travel straight, so we can ignore all directions which drive the drone further from the destination. If the drone cannot find any path, then let the drone consider rest of directions. This will enhance efficiency without reducing quality of the outcome. The search will return a link list when the current node is close to the destination.

2.6 Avoiding Invalid Move

Avoiding a single move entering no-fly-zones is easy, we only need to check for line-to-line intersections, if the move does not enter no-fly-zones, it will not intersect with any edges of no-fly-zones. Also, we check whether the drone is confined by comparing coordinates of where the drone moves to with two coordinates given in page 4 of coursework instruction.

2.7 Back To the Appleton Tower

The drone will return the Appleton Tower if all orders are delivered, or it has low energy. The path back to the Appleton Tower is computed using above algorithm, and the path is calculated for each order. After the food items is delivered, if the remaining battery is sufficient for returning Appleton Tower, the drone will attempt to deliver another order. If the remaining battery is insufficient for delivering the order or returning to the Appleton Tower, the algorithm will cancel the delivered order. Once the order is cancelled, the status of the drone and orders will rewind back to before delivering this order, and the drone will return to the Appleton Tower rather than delivering this order.

2.8 Two Examples

I have chosen 15-01-2022 and 15-11-2023 to render, as they represent both extremes, 15-01-2022 has very small number of orders while 15-11-2023 has large number of orders. Due to limited picture resolution and precision of geojson.io, sometimes the flight path seems like intersecting with no-fly-zones, so I added two zoomed pictures to prove the drone does not fly into no-fly-zones.

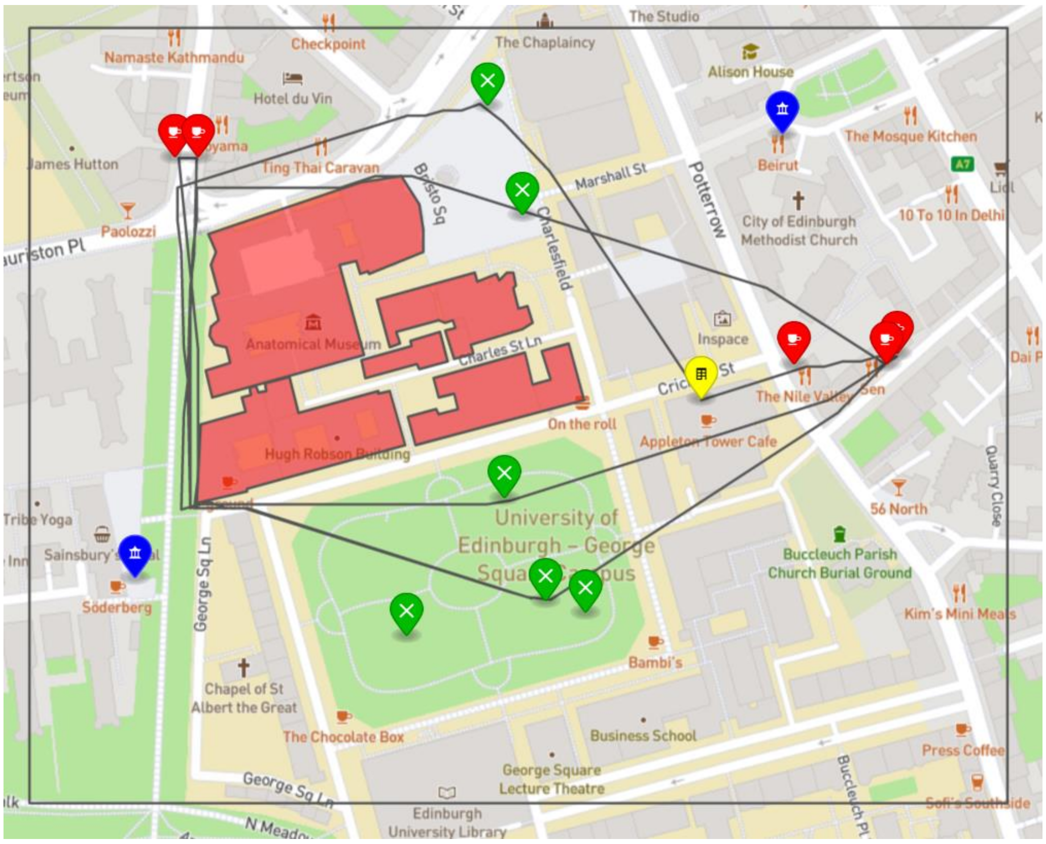
Figure 9: Flight Path 15-01-2022

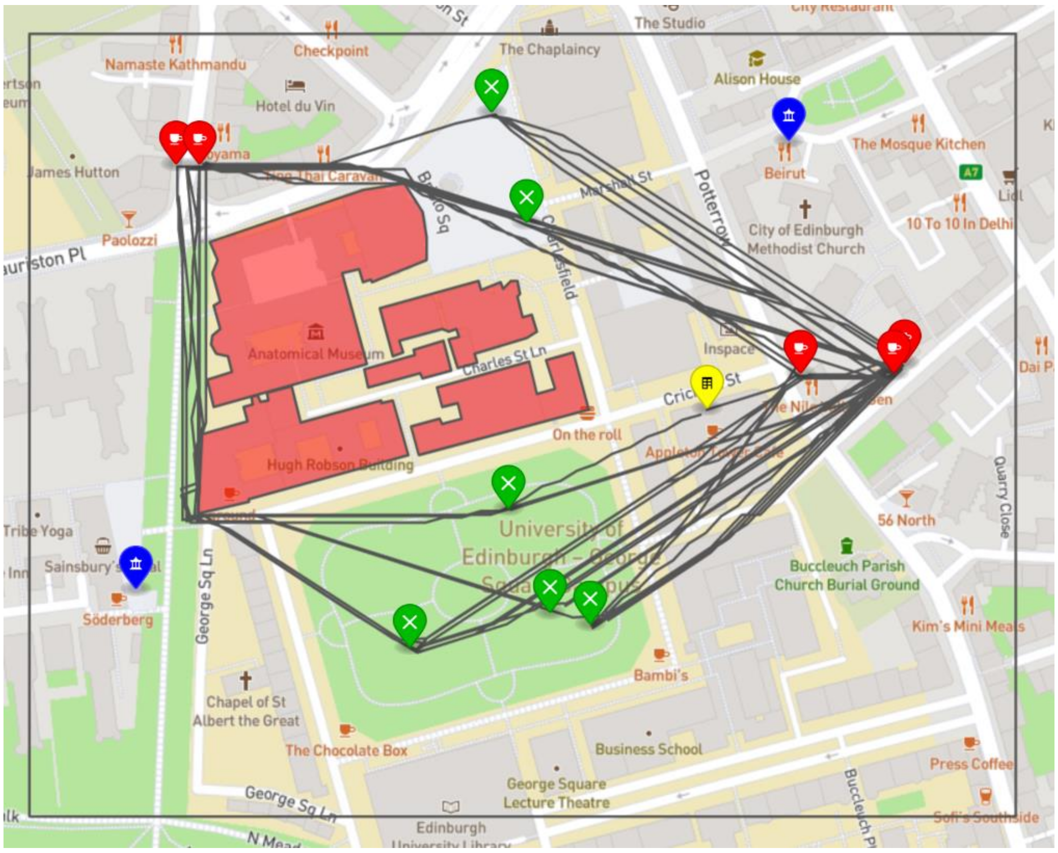
Figure 10: Flight path 15-11-2023

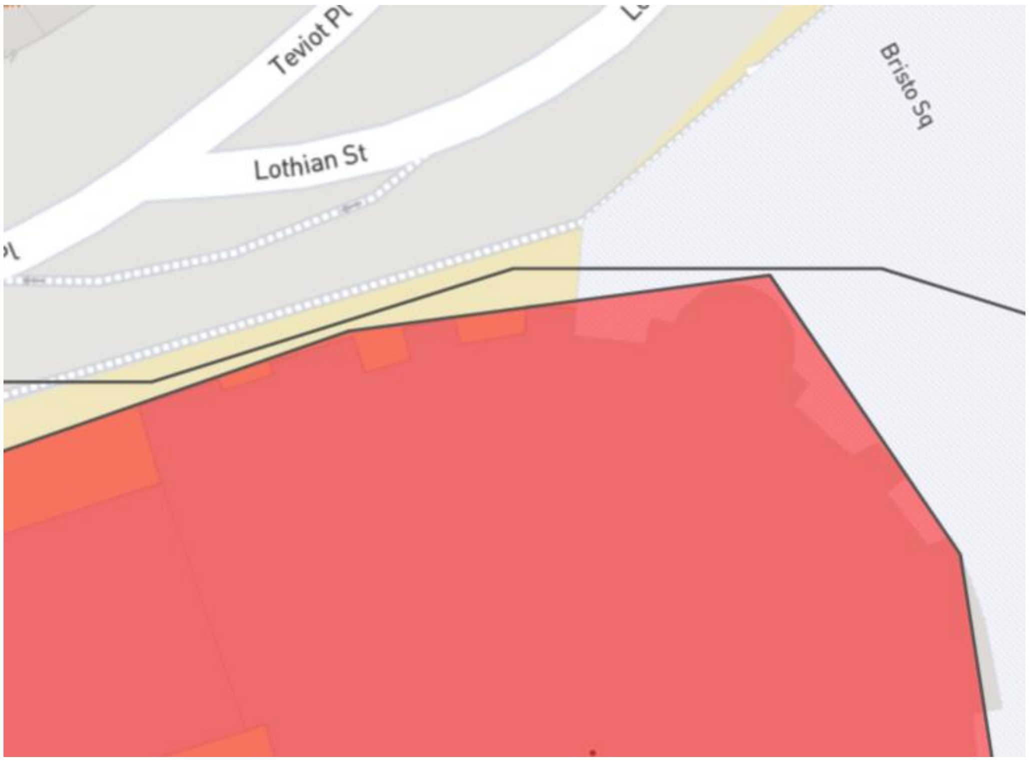

Figure 11: Zoomed in, Top Part of No-Fly-Zones, 15-01-2022

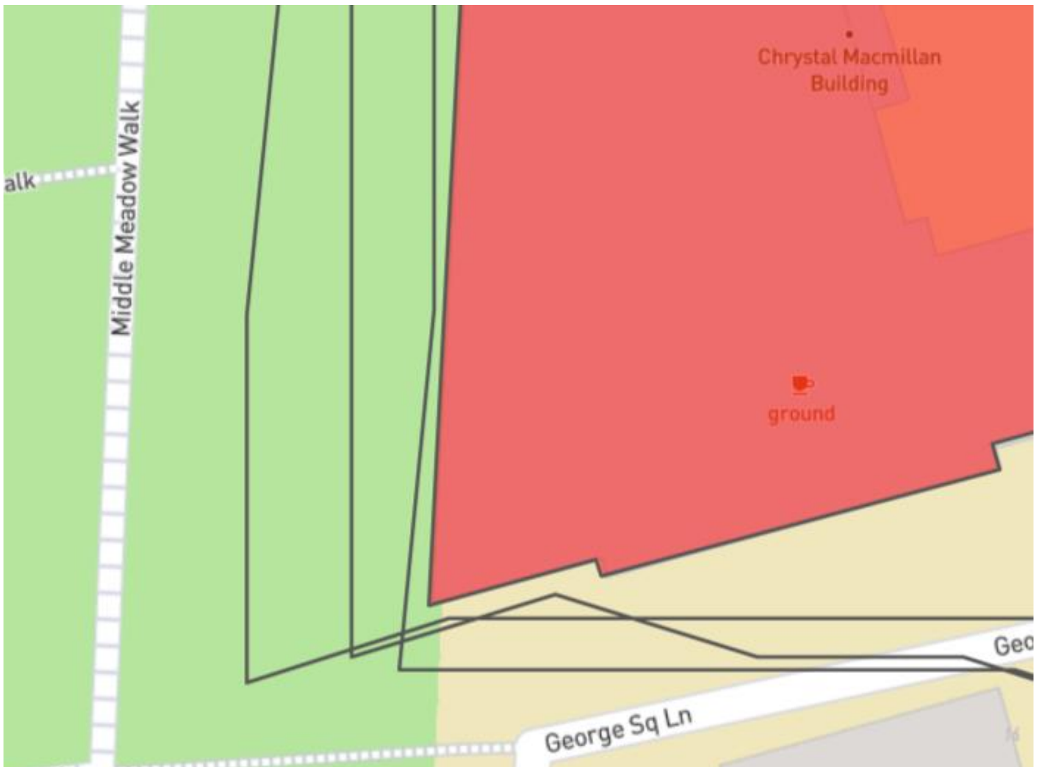

Figure 12: Zoomed in, Bottom Left Part of No-Fly-Zones, 15-01-2022