The University of Edinburgh

The School of Informatics

ILP: Drone Delivery System

s2000527

December 3, 2021

1 Introduction

This report documents the design and implementation of a software system for controlling the flight of an autonomous drone delivering food orders to students at the University of Edinburgh during their lunch break. The system is designed to take into account the constraints on the movement of the drone, constraints imposed by the need to avoid buildings in a no-fly zone and the need to deliver orders one at a time.

2 Background

The School of Informatics is considering a service where students can place orders for food items and drinks to be delivered to them during their lunch break. The orders will be delivered by an autonomous airborne drone which will travel to participating restaurants in the University Central Area to collect the items in a lunch order, and then fly to a chosen location to deliver these to the student who placed the order. It is hoped that this service will allow students to make the most of their lunch break because they will not need to queue at the shops and collect their lunch in person. In addition, the service could be helpful to new students who have just joined the School and have not yet got their bearings and do not know the restaurants near the Central Area.

3 System Overview

The system broadly comprises of, and organises the interaction between 6 main components listed below:

- A database of lunch orders. This database contains information about the order, including the items ordered and the delivery location.
- A website containing information about the participating sandwich shops and their menus, as well as the location of the drop-off points for the deliveries.
- An autonomous airborne drone to carry out order delivery.
- A controller software application which reads information from the database and the website and uses it to control the flight of the drone
- Restaurants: The system uses information about the participating restaurants from the website. This information includes the name of the shop, the address, and the menu.
- Delivery locations: The delivery locations are specified by the drop-off points in the form of 'What Three Words' addresses stored on the web server. These are the locations where the drone will deliver the food orders to the students.

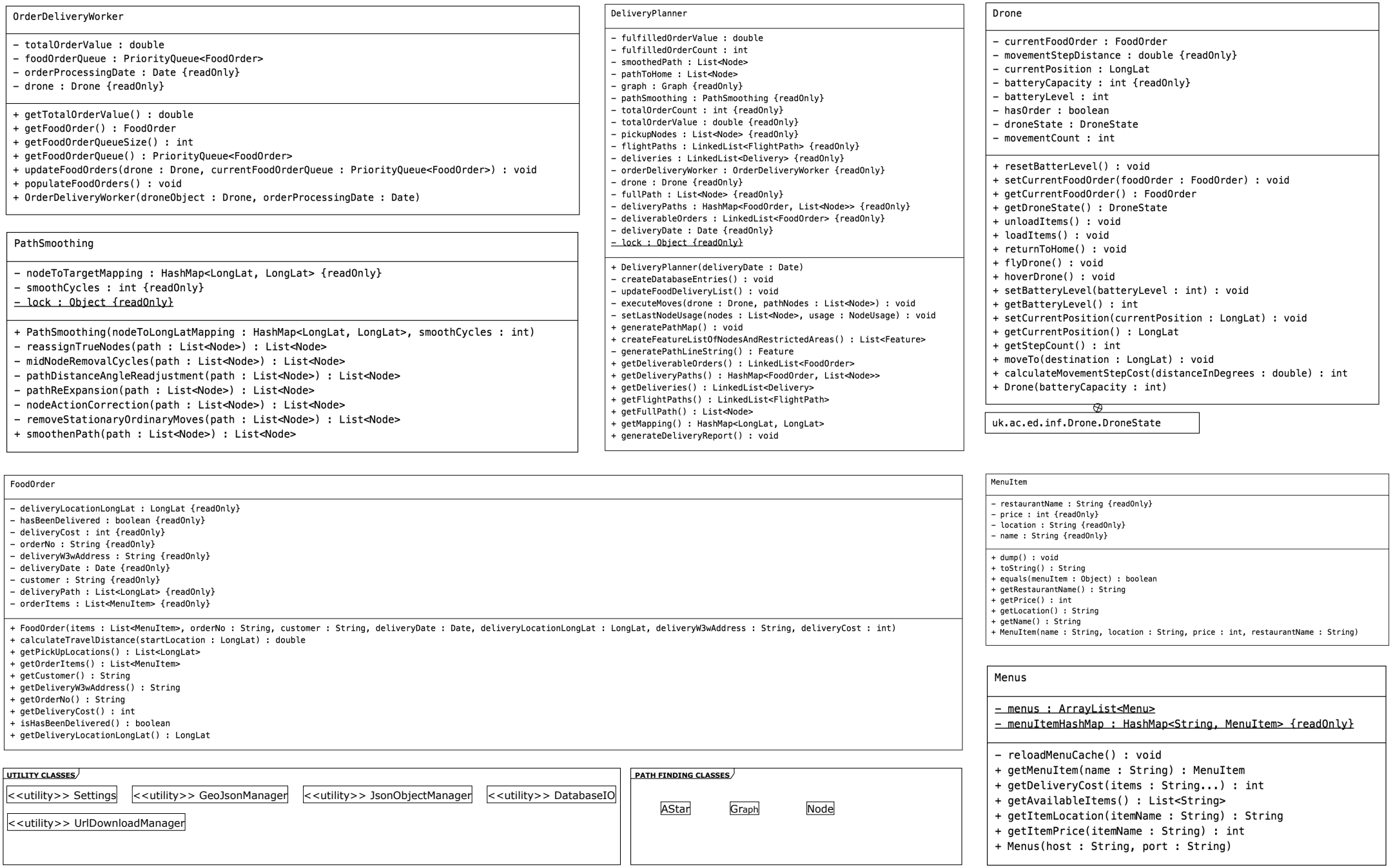
4 Design Constraints

A brief overview of the key design constraints is reiterated below.

- No-fly zones: The drone must avoid flying over buildings in a no-fly zone.
- Limited battery and movement: The drone takes a limited number of steps before the battery runs out. Each step and direction takes the drone a fixed distance in a constrained angle direction.
- Data-driven controller: final solution must be flexible and data driven. This means that the control software reads information from the database and the website and uses it to control the flight of the drone.
- Restaurants: The system uses information about the participating restaurants from the website. This information includes the name of the shop, the address, and the menu.

Software architecture

Adhering to the principles of object oriented design, the system is decomposed into a number of interacting classes to maximise the level of abstraction and future reusability of the system. As primary objects, I Identified the Drone, Delivery Planner and FoodOrder classes as these aggregate all the necessary information to fulfil an order.The overall software architecture is implemented using a collection of java classes as illustrated in the class diagram below. For brevity's sake, only the most integral classes have been expanded.


OrderDeliveryWorker
- totalOrderValue : double
- foodOrderQueue : PriorityQueue<FoodOrder>
- orderProcessingDate : Date {readOnly}
- drone : Drone {readOnly}
+ getTotalOrderValue() : double
+ getFoodOrder() : FoodOrder
+ getFoodOrderQueueSize() : int
+ getFoodOrderQueue() : PriorityQueue<FoodOrder>
+ updateFoodOrders(drone : Drone, currentFoodOrderQueue : PriorityQueue<FoodOrder>) : void
+ populateFoodOrders() : void
+ OrderDeliveryWorker(droneObject : Drone, orderProcessingDate : Date)
PathSmoothing
- nodeToTargetMapping : HashMap<LongLat, LongLat> {readOnly}
- smoothCycles : int {readOnly}
- lock : Object {readOnly}
+ PathSmoothing(nodeToLongLatMapping : HashMap<LongLat, LongLat>, smoothCycles : int)
- reassignTrueNodes(path : List<Node>) : List<Node>
- midNodeRemovalCycles(path : List<Node>) : List<Node>
- pathDistanceAngleReadjustment(path : List<Node>) : List<Node>
- pathReExpansion(path : List<Node>) : List<Node>
- nodeActionCorrection(path : List<Node>) : List<Node>
- removeStationaryOrdinaryMoves(path : List<Node>) : List<Node>
+ smoothenPath(path : List<Node>) : List<Node>
DeliveryPlanner
- fulfilledOrderValue : double
- fulfilledOrderCount : int
- smoothedPath : List<Node>
- pathToHome : List<Node>
- graph : Graph {readOnly}
- pathSmoothing : PathSmoothing {readOnly}
- totalOrderCount : int {readOnly}
- totalOrderValue : double {readOnly}
- pickupNodes : List<Node> {readOnly}
- flightPaths : LinkedList<FlightPath> {readOnly}
- deliveries : LinkedList<Delivery> {readOnly}
- orderDeliveryWorker : OrderDeliveryWorker {readOnly}
- drone : Drone {readOnly}
- fullPath : List<Node> {readOnly}
- deliveryPaths : HashMap<FoodOrder, List<Node>> {readOnly}
- deliverableOrders : LinkedList<FoodOrder> {readOnly}
- deliveryDate : Date {readOnly}
- lock : Object {readOnly}
+ DeliveryPlanner(deliveryDate : Date)
- createDatabaseEntries() : void
- updateFoodDeliveryList() : void
- executeMoves(drone : Drone, pathNodes : List<Node>) : void
- setLastNodeUsage(nodes : List<Node>, usage : NodeUsage) : void
+ generatePathMap() : void
+ createFeatureListOfNodesAndRestrictedAreas() : List<Feature>
- generatePathLineString() : Feature
+ getDeliverableOrders() : LinkedList<FoodOrder>
+ getDeliveryPaths() : HashMap<FoodOrder, List<Node>>
+ getDeliveries() : LinkedList<Delivery>
+ getFlightPaths() : LinkedList<FlightPath>
+ getFullPath() : List<Node>
+ getMapping() : HashMap<LongLat, LongLat>
+ generateDeliveryReport() : void
Drone
- currentFoodOrder : FoodOrder
- movementStepDistance : double {readOnly}
- currentPosition : LongLat
- batteryCapacity : int {readOnly}
- batteryLevel : int
- hasOrder : boolean
- droneState : DroneState
- movementCount : int
+ resetBatterLevel() : void
+ setCurrentFoodOrder(foodOrder : FoodOrder) : void
+ getCurrentFoodOrder() : FoodOrder
+ getDroneState() : DroneState
+ unloadItems() : void
+ loadItems() : void
+ returnToHome() : void
+ flyDrone() : void
+ hoverDrone() : void
+ setBatteryLevel(batteryLevel : int) : void
+ getBatteryLevel() : int
+ setCurrentPosition(currentPosition : LongLat) : void
+ getCurrentPosition() : LongLat
+ getStepCount() : int
+ moveTo(destination : LongLat) : void
+ calculateMovementStepCost(distanceInDegrees : double) : int
+ Drone(batteryCapacity : int)
uk.ac.ed.inf.Drone.DroneState
FoodOrder
- deliveryLocationLongLat : LongLat {readOnly}
- hasBeenDelivered : boolean {readOnly}
- deliveryCost : int {readOnly}
- orderNo : String {readOnly}
- deliveryW3wAddress : String {readOnly}
- deliveryDate : Date {readOnly}
- customer : String {readOnly}
- deliveryPath : List<LongLat> {readOnly}
- orderItems : List<MenuItem> {readOnly}
+ FoodOrder(items : List<MenuItem>, orderNo : String, customer : String, deliveryDate : Date, deliveryLocationLongLat : LongLat, deliveryW3wAddress : String, deliveryCost : int)
+ calculateTravelDistance(startLocation : LongLat) : double
+ getPickUpLocations() : List<LongLat>
+ getOrderItems() : List<MenuItem>
+ getCustomer() : String
+ getDeliveryW3wAddress() : String
+ getOrderNo() : String
+ getDeliveryCost() : int
+ isHasBeenDelivered() : boolean
+ getDeliveryLocationLongLat() : LongLat
MenuItem
- restaurantName : String {readOnly}
- price : int {readOnly}
- location : String {readOnly}
- name : String {readOnly}
+ dump() : void
+ toString() : String
+ equals(menuItem : Object) : boolean
+ getRestaurantName() : String
+ getPrice() : int
+ getLocation() : String
+ getName() : String
+ MenuItem(name : String, location : String, price : int, restaurantName : String)
Menus
- menus : ArrayList<Menu>
- menuItemHashMap : HashMap<String, MenuItem> {readOnly}
- reloadMenuCache() : void
+ getMenuItem(name : String) : MenuItem
+ getDeliveryCost(items : String...) : int
+ getAvailableItems() : List<String>
+ getItemLocation(itemName : String) : String
+ getItemPrice(itemName : String) : int
+ Menus(host : String, port : String)
UTILITY CLASSES
<<utility>> Settings
<<utility>> GeoJsonManager
<<utility>> JsonObjectManager
<<utility>> DatabaseIO
<<utility>> UrlDownloadManager
PATH FINDING CLASSES
AStar
Graph
Node

4.1 Classes

- **Drone:** The drone is best treated as an object on it's own as it has a separate set of attributes and functions. We could additionally also have multiple drones and representing them as individual objects makes this easier.

- **FoodOrder:** We could have multiple food orders each with different states and attributes. An object oriented approach to representing this better reflects that nature of a food order. Relevant identifying attributes have been identified and are reflected in the class diagram.

- **DeliveryPlanner:** Majority of the classes aim to maximise abstraction across the whole project. The delivery planner class is meant to manage the multiple interactions between these classes. It's ties everything together and manages move planning, execution and data logging.

- **OrderDeliveryWorker:** An essential component of order delivery is deciding the order in which we deliver the orders. The order delivery worker manages this task by creating and updating a priority queue based on the drone's location.

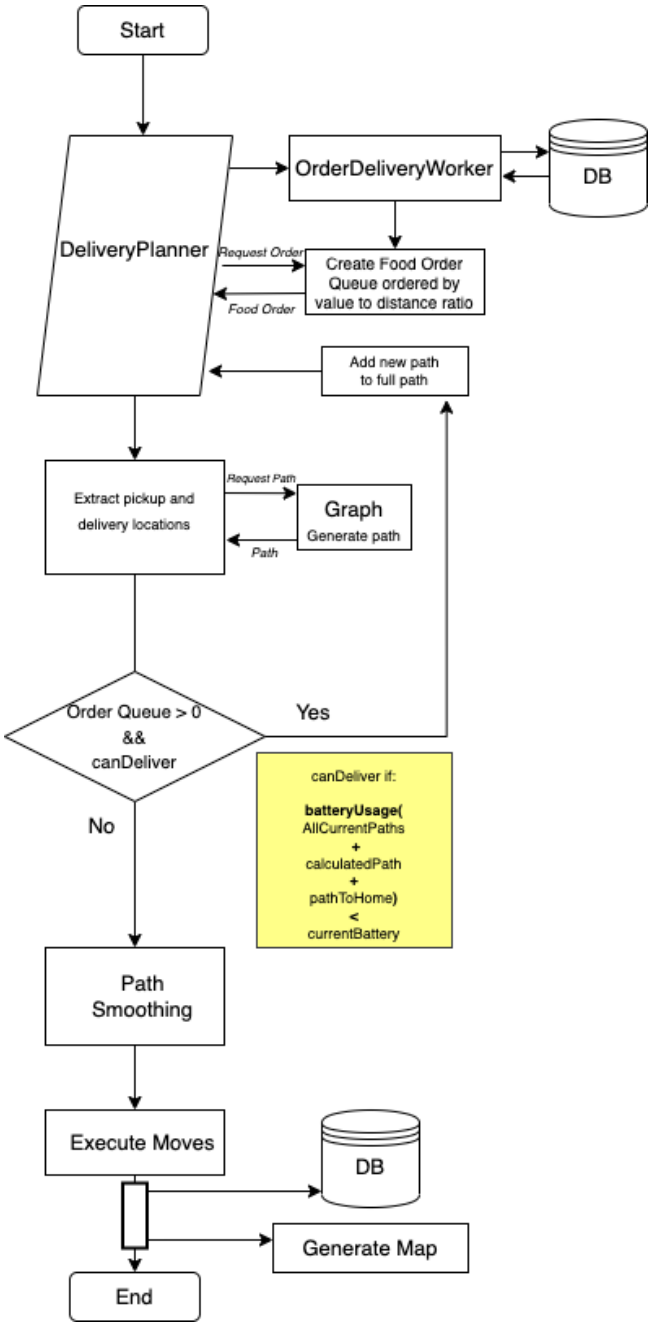
- **Utility Classes:** Not all classes necessarily need to be treated as objects. Some classes only perform what I would term 'handshake' interactions in which no knowledge about the past state is required. Some of these functions include, keeping track of variables, writing data to the database, loading and parsing specific server files etc. These are managed by a set of utility classes as shown in the diagram.

- **Path finding classes:** As would be used in a typical path finding algorithm implementation, we have classes to represent 1) The algorithms used('AStar' in this case), 2) A search space('Graph' in this case) and 3) Nodes in the graph('Node' in this case).

- **Path Smoothing:** While initially seeming somewhat related to our path finding classes, Path Smoothing performs a rather different class of operations on nodes. Path smoothing is performing what would be considered a post processing operation which includes relabeling nodes in the given list. They are in grouped in the same package but have been abstracted to different classes to reflect this difference.

4.2 Data flow

An illustration of the flow of data when the algorithm is running is shown below.



5 Drone Control Algorithm

A deterministic approach is taken in my implementation reason being the simplicity of implementation and faster path finding. However, this can also be done using a non-deterministic algorithm if needed. Reasons for taking this approach include:

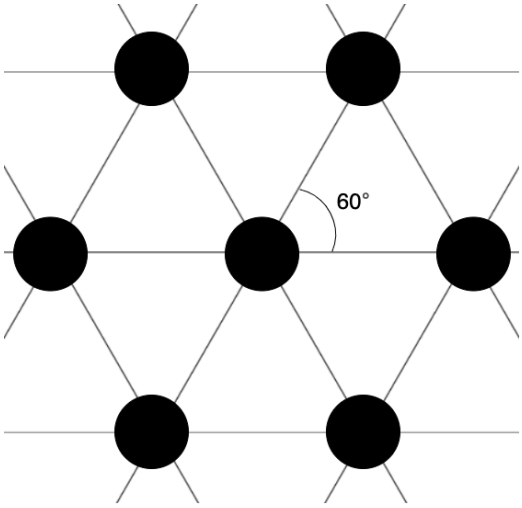
- The number of nodes that need to be generated and evaluated is considerably lower than that of a non-deterministic algorithm[1].
- While this(nodes generated) can be reduced considerably if some heuristic is employed e.g. straight line heuristic, the existence of restricted areas within the search space will lead to a number of tricky edge cases leading to bad worst case performance or the failure to find a solution.

- In the case of a services as essential as a delivery service, having a deterministic algorithm is of the advantages as we guarantee the drone will always perform consistently as opposed to a non-deterministic case where we could possibly have unexpected behaviour.

  Note: While we have a fixed set of delivery and pickup locations, the approach taken in this design aims to guarantee the delivery orders to **ANY** location provided it is a valid location(within confinement area and not in a no-fly zone).

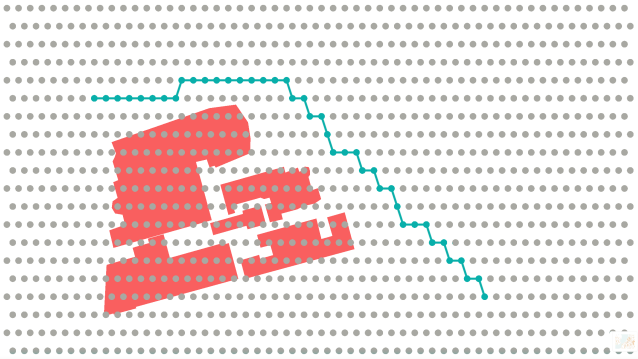
5.1 Grid representation

The search space is represented by a 2 dimensional triangular mesh with each node equidistant to its adjacent nodes. This distance has been set to the default drone step distance of 0.00015 degrees which guarantees us that the drone is able to get from each node to the next within a single step. Further, as this is a triangular mesh, the angle between any two adjacent nodes is always a multiple of 10. Lastly a slight margin, the size of half a single step is added to each edge of the grid to prevent nodes from being generated on the boundary of the restricted area. This is illustrated in the following diagram.



5.2 A star algorithm

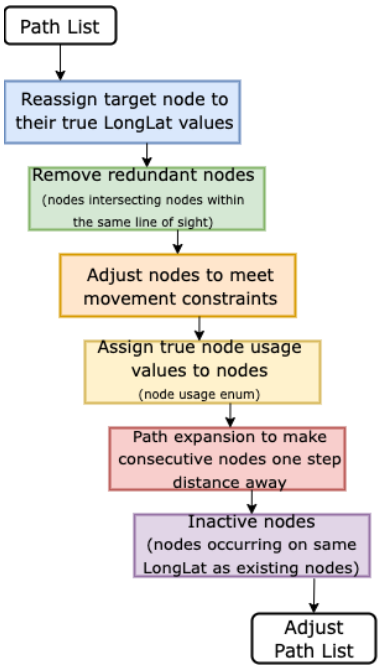
As an industry standard in cases where the straight line heuristic is applicable , I employed the use the AStar[2] path finding algorithm. This was chosen as it is a well-known and reliable algorithm that is capable of dealing with obstacles. The path finding algorithm is implemented as a java class called AStar. Typical modifications were made to the algorithm to take into account the specific constraints of our system. These include a fixed step cost between valid nodes while and infinite step cost on transitions from valid to invalid nodes. This is done because the our original generated grid spans the entire map area and we know for a fact that the map may contain Nodes that fall within restricted areas. When applied in its base form, the algorithm produces as path similar to what is illustrated below.



5.3 Post smoothing

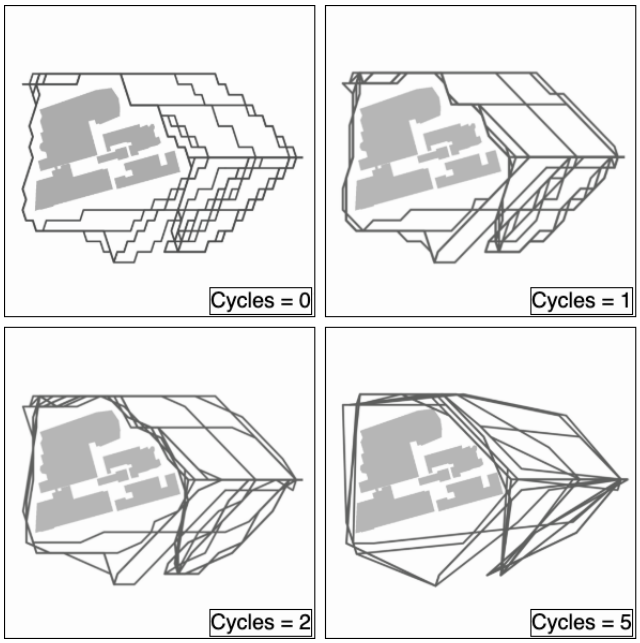
Due the nature of our search space, it is possible that the generated path from the restaurant to the delivery point is not smooth. What is meant by this is that often generated paths follow a zigzag pattern as that is how the nodes are spaced. While this gets from the start to the delivery location, it is often inefficient especially in the case of multiple deliveries as will be illustrated below.

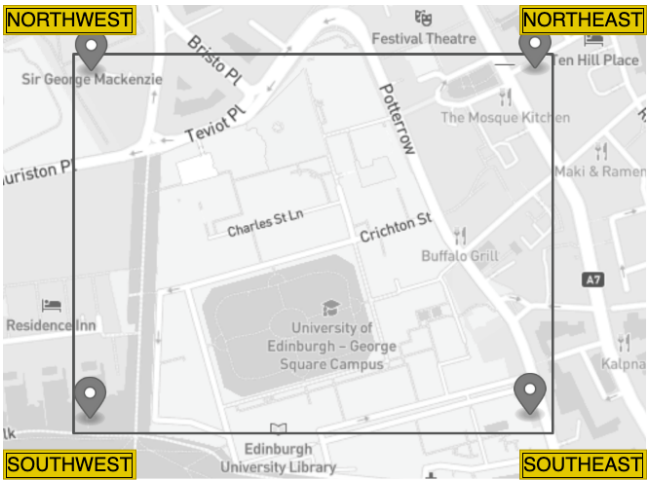
While simply using a simple line of sight check between nodes is popular approach[3] to dealing with this and can help smoothen out the desired path by getting rid of redundant nodes, the movement constraints make this a lot more complicated. An algorithm that implements a pipeline of 6 path processing layers has been employed to generate a path that adheres to the desired movement constraints and is illustrated below.



5.4 Effect of Path Smoothing

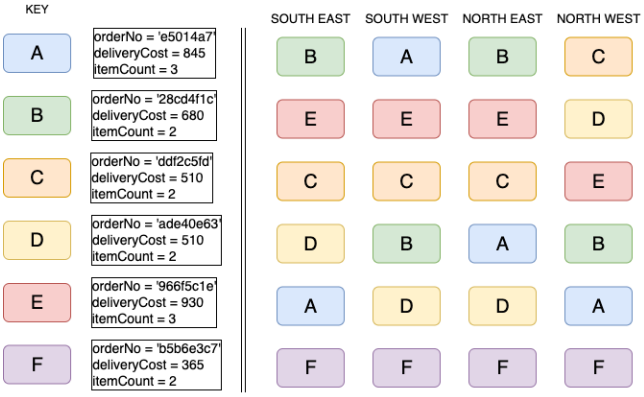
An additional *path_ smoothing_ threshold* variable has been added as a way to control the granularity of the smoothed output path. The initial effect of smoothing on a path is that it eliminates redundant nodes. The output of this is then used to correct for angle and movement constraints. If too many nodes are extracted, we may not have enough flexible nodes that can be used to correct for distance and movement angles hence we have to minimise how many times we cycle through the path and extract redundant nodes. An illustration of applying more node extraction cycles is shown below.





For a collection of six randomly selected food orders conveniently named A,B,C,D,E,F the ordering of items in the priority queue as the drone position is switched across the four corners is shown below.

Heuristic based reordering of orders



Considering the relatively low value of item 'F', it is no surprise that it has the lowest priority from all 4 locations.

5.5 Order Queuing System

As an additional measure to maximise the optimality of our final algorithm, we have employed the use of a heuristic based order queuing system[4]. Orders are stored in a priority queue and are orders by their cost to travel distance ratio. Denser food orders(ones with higher value per distance traveled) are prioritised. This distance measure is based on the drones current location, locations of shops that will need to be visited and including getting the order to the customer. This is not a exact method as it approximates this with the assumption that locations are connected by straight lines. Presence of restricted areas prevents this from holding. An illustration of item reordering based on the drones location is shown below.

6 Full Rendered Flight Paths

The diagrams below show renderings of the full flight path on two selected dates.

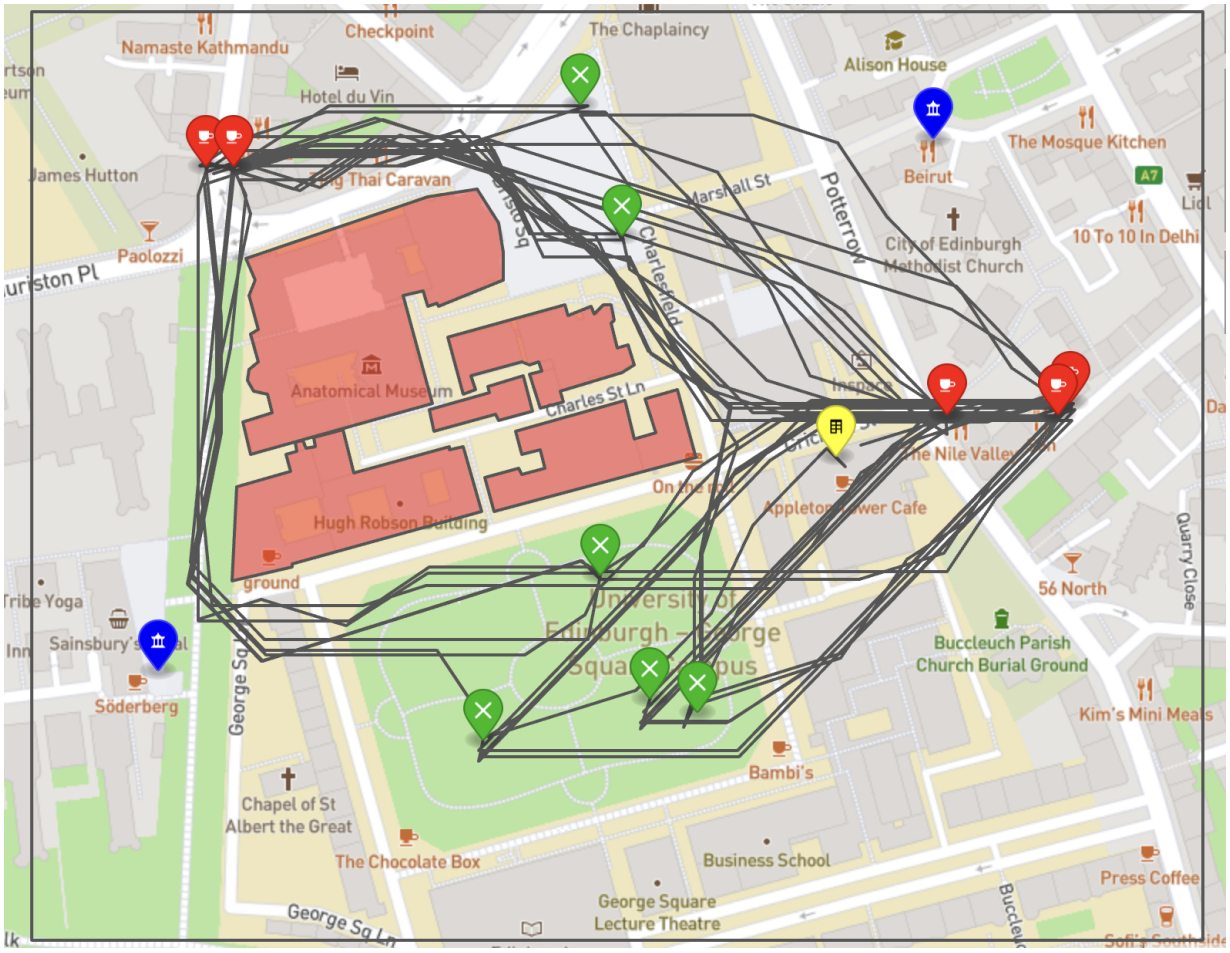


Figure 1: 14/08/2023

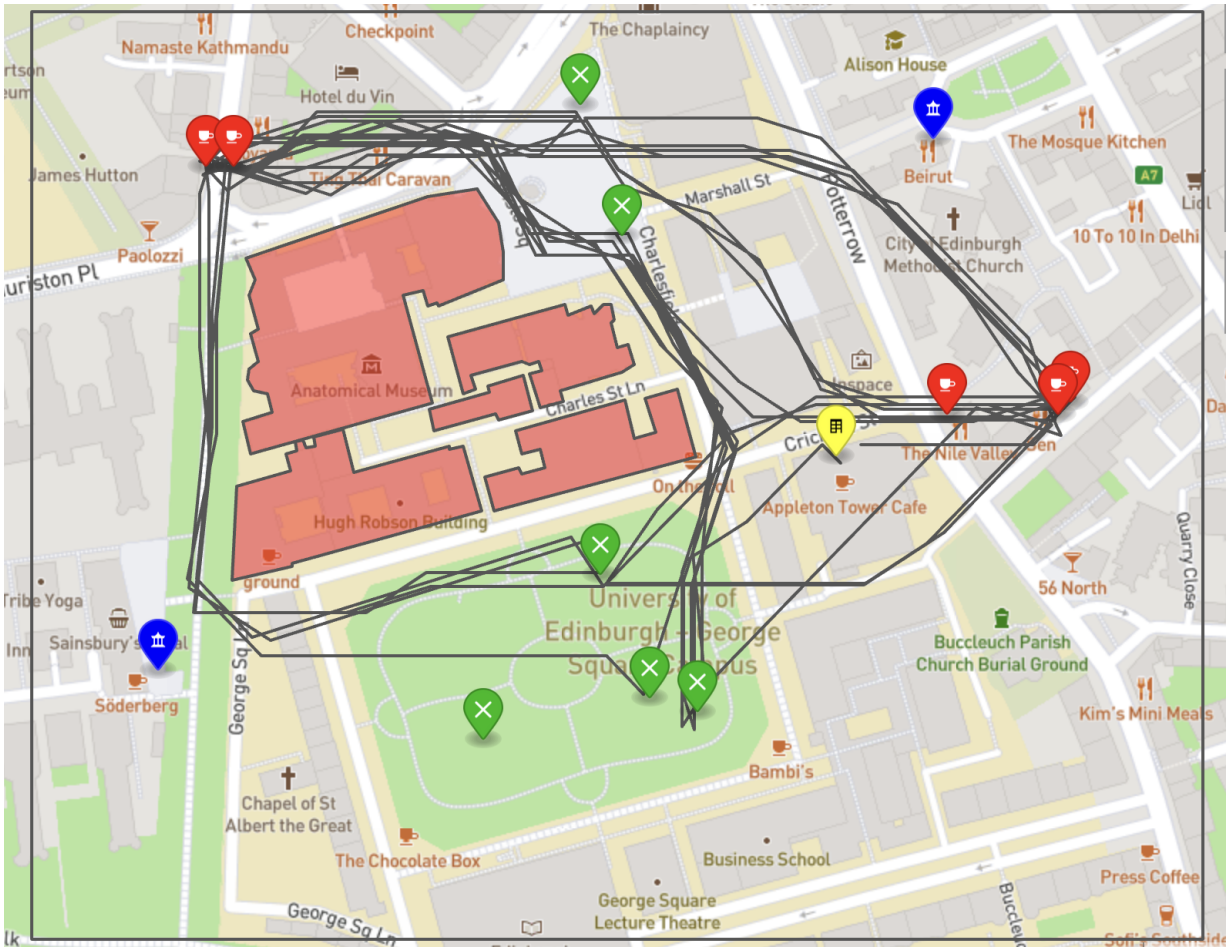
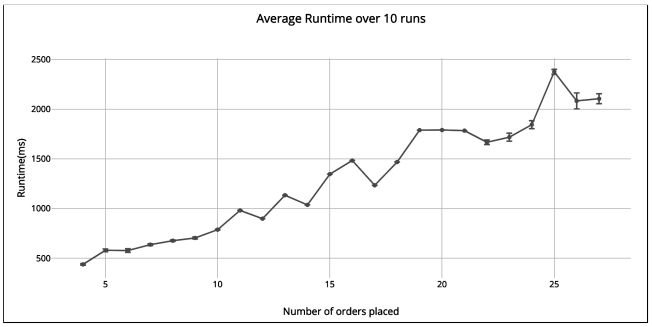


Figure 2: 30/11/2022

7 Testing and Evaluation

Testing and evaluation of the software was done to assess how the execution time scales as the number of orders increases. I ran 10 orders for each order count where the count is the number of orders made on a specific day. The lowest order count was 4 while the highest was 27 and each of these was used to compute the mean and standard deviation. A line chart is shown below to illustrate runtime including bars for the standard deviation.



This is clearly a rather simplistic assessment of the performance as we are assuming the average travel distances should be roughly that same for each day. Despite this assumption, it should still give us a good idea of how the run time might scale as we increase the number of orders. In this case, we have what seems to be linear growth(O(n)) with the size of the input.

Further, the sampled monetary value was computed for each delivery day but this has not been included in this report. The reason for this exclusion is that we had obtained a value of 100% for all our orders and no further break down could be given to account for any anomalies as there were non.

8 Open questions

- While using the straight line heuristic to order our food orders is sufficient for the most part, it does not give us the optimal order. We could get closer to this optimal ordering by considering the actual travel travel distance to deliver and order. This could be done by modifying our current system to calculate the travel path from the initial restaurant all the way to the final destination. This could stored in a hash map which we could use to find the travel distance and reorder our queue based on this.

- Having an algorithm that simply prioritises students that are willing to spend more money does not seem inclusive. Simply prioritising higher value items has tendency of completely ignoring lower value orders even if they were placed before the higher value orders. It would be a good idea to use some weighted value that involves considering how long an order has been in the queue. This should add some weight to orders that have been in the queue longer and thus they wont be ignored completely.

9 Conclusion

This report documents the design and implementation of a software system for controlling the flight of an autonomous drone delivering food orders to students at the University of Edinburgh during their lunch break. The system is designed to take into account the constraints on the movement of the drone, constraints imposed by the need to avoid buildings in a no-fly zone and the need to deliver orders one at a time. Further, the system has been designed to fulfill an order placed in any location provided it is a valid location. The system has been implemented and tested using simulated orders. The results of the testing show that the system is able to plan a flight that guarantees the delivery of orders within a reasonable amount of time. The system also appears to be able to handle the number of orders placed during the lunch break period with a minimal increase in the run time of the system as the number of orders increases.

References

[1] Jacques Cohen. Non-deterministic algorithms. *ACM Computing Surveys (CSUR)*, 11(2):79–94, 1979.

[2] Wikipedia. A* search algorithm — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=A*%20search%20algorithm&oldid=1055876705, 2021. [Online; accessed 02-December-2021].

[3] Konstantin Yakovlev, Egor Baskin, and Ivan Hramoin. Grid-based angle-constrained path planning. In *Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)*, pages 208–221. Springer, 2015.

[4] Dave Ferguson, Maxim Likhachev, and Anthony Stentz. A guide to heuristic-based path planning. In *Proceedings of the international workshop on planning under uncertainty for autonomous systems, international conference on automated planning and scheduling (ICAPS)*, pages 9–18, 2005.