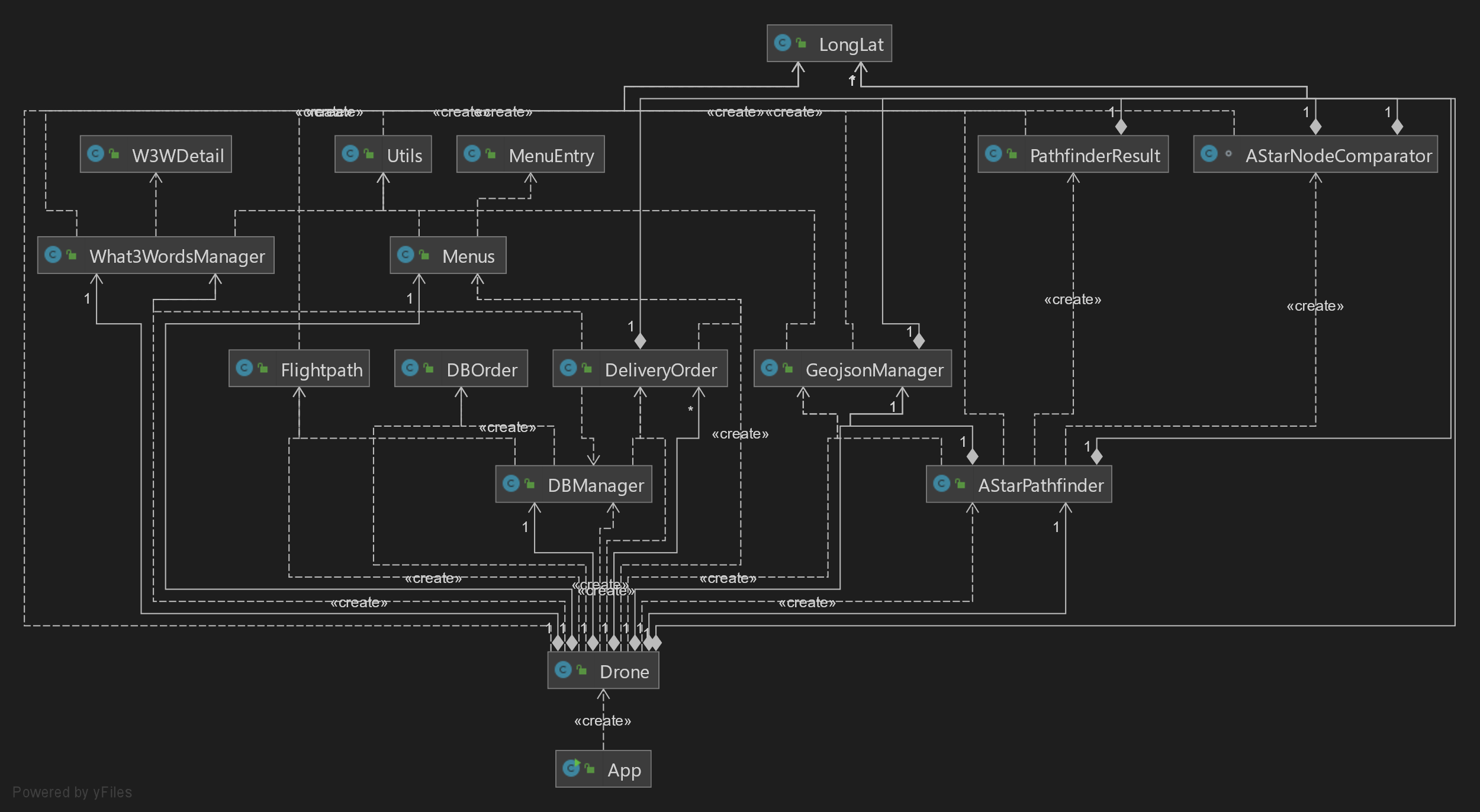
ILP Report

# Software architecture description

The software is implemented in Java and consists of the following Java classes: ***App*, *Drone*, *AStarPathfinder, DBManager, GeojsonManager, What3WordsManager, Menus, DBOrder, LongLat, Flightpath, DeliveryOrder, MenuEntry, W3WDetail, PathfinderResult, AStarNodeComparator, Utils****.* That are 16 top-level classes in total, some of which has inner classes used for marshalling server responses or database records, which are not of much significance on their own since they do not interact with other class instances, and act only as data holders. It is worth mentioning that there is a sub-package for A\* pathfinding related tasks, named “***AStarPathfinder***”, which contains the 3 classes ***AStarPathfinder,*** ***AStarNodeComparator*** and ***PathfinderResult.*** A hierarchical UML diagram is as follows:



As can be seen from the UML diagram, the entry point is in class ***App***, which is what every Java program needs to be started. Within the entry point it creates a ***Drone***object, representing the delivery drone as its name suggests. The delivery drone would then be responsible to plan its delivery route and carry out the delivery, which further requires a range of different tasks to be completed. There are some common computations and functions that are needed by a range of tasks, such as sending HTTP requests and performing mathematical calculations of determining if any two line segments intersect with each other, which are placed in the class ***Utils***. This class is never instantiated, and all member fields and methods are static.

It is apparent that to implement a drone delivery service, the drone itself will need to be modelled, therefore the class of utmost importance is ***Drone.*** As per the specification and requirements, there will only be 1 instance throughout the lifetime of program execution, and it is clear that any other class instances would be associated to it, which can be seen from the above UML diagram. Also obviously needed is a representation of geo location, since the software works with geographical information and involves movements. Therefore, the ***LongLat*** class is identified and created to represent locations, as well as to support all computation needed on geographical movements. To realise the delivery functionalities, orders from students/customers will need to be fetched from the database. Furthermore, it is required that we write data of our drone’s delivery and flightpath into the database. This naturally calls for the class ***DBManager***, which will handle the reading and writing of database tables and records. An instance will be created by the drone as its member, to facilitate database access. It is then clear that we need to marshal the database records, specifically, the records from table “*orders*”, which then leads to the class ***DBOrder*** being identified and created. Another need of the drone is that it requires access to data on menus and items, therefore similarly, the class ***MenuEntry*** and ***Menus*** can be identified and created, which are responsible for marshalling data from the webserver and providing information on the menu items and costs, etc. In order to perform deliveries, the drone will also need exact longitude and latitude from the what3word encoding. This leads to the identification of class ***W3WDetail*** and ***What3WordsManager***, which are responsible for data marshalling and information retrieving, in this case, retrieving the longitude and latitude encoded by what3word. Although this class has few functionalities and could be considered part of the pathfinding task, it is important and meaningful that we reduce coupling and improve modularisation by identifying and creating these 2 separate classes. There is still one more task of data marshalling and processing needed of the drone, which is the operating with Geojson files, since we are asked to both read from and write data to them. Again, similar to the aforementioned cases, the class ***GeojsonManager*** was identified and created, which is responsible for retrieving, marshalling and processing the data from the webserver, as well as doing geo-computations such as no-fly-zone detection which depends on Geojson data. It is worth mentioning that these manager classes can be designed to work as singleton objects due to the characteristics of the coursework specifications and requirements, for example, database connection and operation. However, a design choice was made not to, because the singleton pattern is arguably, an anti-pattern “because it often introduces unnecessary restrictions in situations where a singleton class would not be beneficial, thereby introducing global state into an application. Additionally, when a certain class interacts with a singleton, that class and the singleton become tightly coupled”. (Wikipedia, 2021) Instead, it was made sure that throughout the whole program, the one and only ***Drone*** instance only ever instantiate one instance of such manager classes, and pass them around when needed by other class instances, i.e., the manager classes. Another main problem, functionality wise, is pathfinding, which is clearly needed by a delivery drone. Again, to reduce coupling and help with easier testing, the class ***AStarPathfinder*** is identified, under the package with the same name, which will provide the pathfinding functionality for the drone. After further exploring and developing, I realised that two extra (helper) classes are needed: ***AStarNodeComparator*** and ***PathfinderResult***, which are respectively used to implement the core part of A\* algorithm of comparing pathfinding nodes, and to encapsulate the pathfinding results of distance/cost and actual path found which will be needed by the drone. Finally, classes are identified with the progress of development, and as a result are more implementation dependent. Classes ***DeliveryOrder*** and ***Flightpath*** are identified during development, for the purpose of encapsulating data for a delivery order for the drone, and the movement/flightpath of the drone, which needs to be written to database. The key difference between a ***DeliveryOrder*** and a ***DBOrder*** is that the latter pertains directly to the database record, whereas the former holds more data, in fact, all the data needed for the drone to plan for and carry out deliveries.

# Drone control algorithm

It is expected that the drone control algorithm will maximise the drone’s score on the *sampled average percentage monetary value* metric, which is the total monetary value of deliveries made divided by the total monetary value of orders placed. Simply put, the drone should behave in way that aims to make deliveries that at the end of the day, makes the most money, under the limited number of moves it can perform each day. This would mean that first, the drone should perform movements efficiently so as not to waste the limited moves, thereby being potentially capable of delivering more orders; and second, it should make smart plans about which orders to deliver (or rather, which not to) when it turns out that it is impossible to deliver all of the orders for the day. This may be generalised as a pathfinding problem for the former, and a Travelling Thief Problem for the latter, which “is a challenging combinatorial optimization problem. The TTP interconnects two well-known NP-hard problems: the Travelling Salesman Problem (TSP) and the 0-1 Knapsack Problem (KP).” (Lei Yang, 2020).

For the pathfinding problem, I decided to use the A\* pathfinding algorithm, which is a graph traversal and path search algorithm, which is often used in many fields of computer science due to its completeness, optimality, and optimal efficiency. (Wikipedia, 2021) It suits the problem at hand, but there is a problem I anticipated. The drone can move in 36 different angles, and that it is in a continuous space instead of in a grid-based or graph one, which I believe means the graph to traverse is potentially huge, since it would be like trying to represent a continuous space with discrete nodes. There is another problem which I think may complicate the A\* pathfinding even more and subsequently lengthens the program’s runtime, that is the rounding error, which could perhaps result in graph nodes that are supposed to be the same being different. However, these are just my unverified guesses. But to avoid potential problems, one possible solution is to turn the original continuous space into a discrete one, for example I can simply turn the problem at hand back into the classic grid map pathfinding, which suits A\* algorithm perfectly. But this raises another problem, that is the drone will be travelling in longer routes all the time. The final solution I came up is to use a Visibility Graph, where “Each node in the graph represents a point location, and each edge represents a visible connection between them.” (Wikipedia, 2021) Or alternatively, a polygonal map, which is a “common alternative to grids […]. If the movement cost across large areas is uniform, and if your units can move in straight lines instead of following a grid.” (Patel, 2021) What my pathfinder does is it puts together a graph for the A\* algorithm by using the vertices of polygons that constitute the no-fly-zone, plus the starting point and goal position, as nodes of the graph. The informal rationale is that, if we can reach the goal from starting point, pathfinding works perfectly fine; if not, then by performing A\* search on the visibility graph, it is guaranteed that a Euclidean shortest path can be found by visiting some of the polygon vertices in straight lines, given the optimality of A\* search and triangle inequality. It is also guaranteed that the drone will not cross the polygons because it will only ever be moving along the perimeters, or between vertices. There is a caveat in our case, which is the drone may not be able to travel along some graph edges, due to its limited moving angle and distance, for example, through tunnel-like terrains where it could hit the walls on either side. To mitigate such issues, during the graph creation phase, nodes that can be reached in a straight line will be checked that it is not too hard to reach for the drone. A piece of pseudocode illustrates the idea of such checks.In summary, my implementation will use A\* search to find a list of waypoints, that is, some of the vertices of the no-fly-zone polygons, and let the drone follow them. A piece of pseudocode would be as follows:

There is still another problem to be solved. The drone cannot completely follow the planned straight-line path/waypoints because it can only travel in fixed angles and move in fixed distance at a time. An extra layer of pathfinding, or path following is needed of the drone’s pathfinding algorithm. For example, when the drone intends to travel along the perimeter of the no-fly-zone, it needs to make sure that it doesn’t go into it (briefly, for one move) due to its stiff moving angles. Therefore, for every move the drone makes, it needs to check that it’s not crossing the border of the no-fly-zone. The drone will rely on the ***GeojsonManager*** to provide information on whether a move will cross the perimeters of no-fly-zone, and choose from moves that does not. The line segment intersection algorithm used is purely mathematical, and borrowed ideas from online (Rees, 2017). The algorithm will then rely on a greedy 2-step exhaustive search for actual drone movements minimising for the straight-line distance between drone’s position and the waypoint it is aiming to reach, while respecting the fact that no move can ever cross the perimeter of the polygons of no-fly-zone. It will stop when it’s close to the said waypoint. The reason why 2-step search is needed is due to the fact that a greedy search with 1 step lookahead can easily get stuck with no solution, i.e., being close to waypoint, that is, a no-fly-zone vertex, and blocked by the perimeter of no-fly-zone, and failing to move past it because a good move will violate its greedy nature. With 2-step greedy, the algorithm has more insight, and is bound to find a solution, due to the fact that being the distance of being close to is the same as the movement distance. An informal justification for this is that since those two distances are the same, with any movement greater than 1 move, the drone is guaranteed to get closer to the goal waypoint, and not be blocked by the no-fly-zone perimeter. A piece of pseudocode is as follows:

In essence, the pathfinding algorithm is a combination of A\* global search and greedy local search, which enables the drone to quite effectively traverse the map.

For the second part, the Travelling Thief Problem, it is in fact quite complex, with various proposed solutions to the problem, such as genetic and evolutionary computation (S. Polyakovskiy, 2014), heuristics based local search (MohamedEl Yafrani, 2015). In the end I decided to design an algorithm that is much simpler to implement by combining the 2-Opt optimisation (Wikipedia, 2021) for Travelling Salesman Problem with a greedy approach based on cost effectiveness (bang-for-buck) for removing orders from the trip when it is impossible to deliver all orders. However, another optimisation is carried out first, that is the order of visiting the items’ pickup locations (for those that have a second pickup location). It is guaranteed to be globally better for the TSP problem, if the local cost (movements needed for completing one delivery) can be improved, since we are treating one full delivery as a node in the TSP problem. This is done quite trivially, by looking at each delivery order individually and swapping its items’ pickup locations order if it turns out better.

Pseudocode for the first two main ones are as follows:

To be more exact, a regular 2-Opt on TSP is performed first (with a capped number of iterations that grows with the number of nodes, to comply with the time constraint). If it happens that we cannot finish delivering all orders, the drone will start dropping orders one by one, until it can complete all deliveries. The decision process of dropping orders is guided by the greedy approach, which in essence removes the order in the current unfinishable trip that has the lowest value of monetary value of order divided by the distance the drone needs to travel. This is apparently not a very smart heuristic, since there is interdependency between choosing of orders and the result of TSP optimisation. Basically, a chicken-and-egg problem for heuristics. However, it yields a performance uplift compared to my previous implementation, which is to perform a greedy optimisation on TSP and a random removal of order when not all orders can be delivered.

The following are 2 Geojson visualisation on date 31-12-2023 and 11-11-2022.

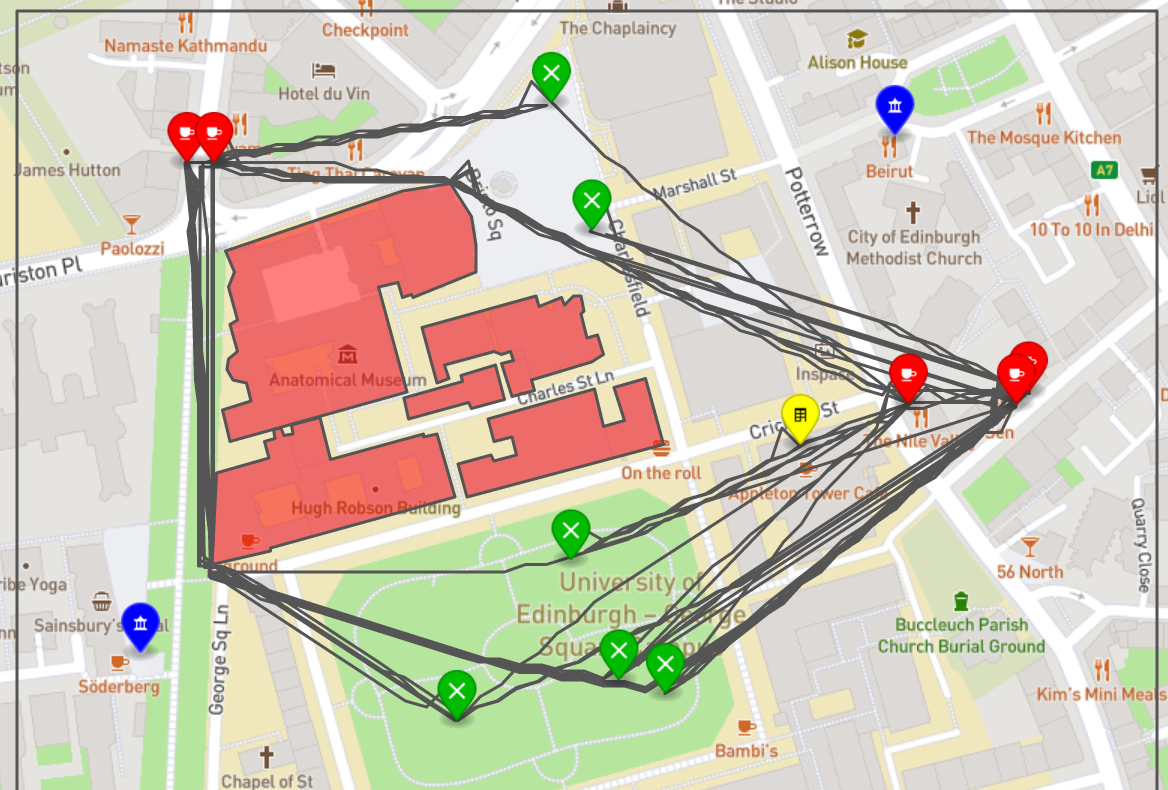


Figure 31-12-2023

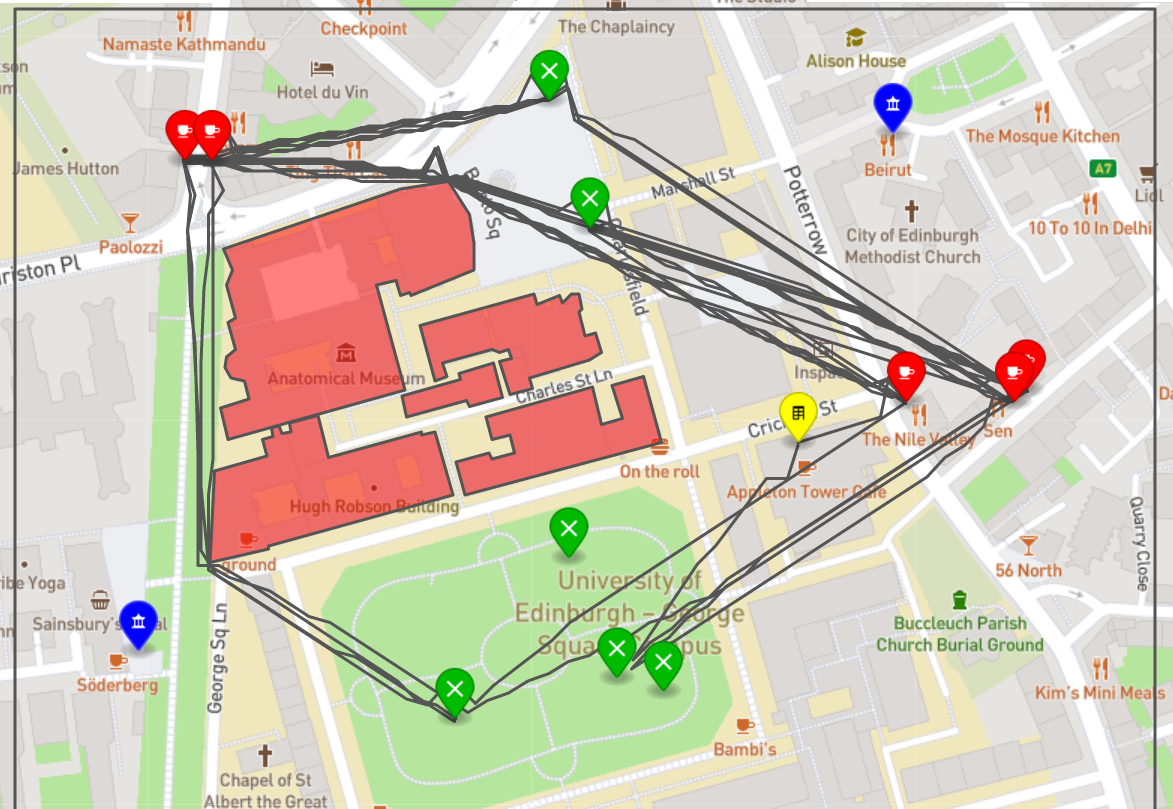


Figure 11-11-2022

It may appear that the drone’s flightpath is crossing the no-fly-zone, but that is due to the zooming level, when zoomed in, it can be seen that no flightpath crosses the polygon perimeters.

The metric of sampled average percentage monetary value appears quite satisfactory. The drone is seemingly able to deliver all orders throughout yar 2022 and the majority of 2023 (from some of the dates I checked against), until it has to drop some orders later in 2023. Which is to say, the sampled average percentage monetary value is 100% for the majority of the given dates, and for the end of the year 2023 which is the period of time with most orders each day, the metric is approximately above 95% for a sampled 7-day period as shown below. It is worth noting that, due to the random nature of 2-opt, the reported metric is not a fixed value, for example, for 27-12-2023, the drone might be able to achieve a metric of 91.9% with still a total of 23 orders delivered; for 30-12-2023, it could achieve a 97.5% with the same number of 24 orders delivered.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Dec 2023 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| Metric | 96.4% | 98.8% | 90.8% | 100% | 97.2% | 96.9% | 99.5% |
| Total | 27 | 27 | 27 | 27 | 27 | 27 | 27 |
| Delivered | 25 | 26 | 23 | 27 | 26 | 24 | 26 |

Overall, the drone control algorithm mainly consists of a pathfinder with A\* search, and a Travelling Thief Problem with a 2-opt route optimisation combined with a greedy Knapsack item picking. The results have looked quite acceptable, as can be seen from the table above.

# References

Lei Yang, Z. Z. (2020, December 16). *Solving the Travelling Thief Problem based on Item Selection Weight and Reverse Order Allocation.* Retrieved from arxiv: https://arxiv.org/abs/2012.08888

MohamedEl Yafrani, B. A. (2015, April 4). *A local search based approach for solving the Travelling Thief Problem: The pros and cons.* Retrieved from Science Direct: https://doi.org/10.1016/j.asoc.2016.09.047

Patel, A. (2021, October 13). *Map representations.* Retrieved from Amit’s Thoughts on Pathfinding: http://theory.stanford.edu/~amitp/GameProgramming/MapRepresentations.html

Rees, G. (2017, August 25). *How do you detect where two line segments intersect? [closed].* Retrieved from Stack Overflow: https://stackoverflow.com/questions/563198/how-do-you-detect-where-two-line-segments-intersect

S. Polyakovskiy, M. R. (2014, July 12). *A comprehensive benchmark set and heuristics for the traveling thief problem.* Retrieved from Google Scholar: https://dl.acm.org/doi/abs/10.1145/2576768.2598249?casa\_token=LnBszOkw1fIAAAAA:jTO-niJONqsUyVGNchSaOPyzB9N-ltUV64okq5GcX4y5i3Oh3SzLMa\_a\_uFiQ-\_IE6zt6sR8BXQ8cQ

Wikipedia. (2021, Decmber 1). *2-opt.* Retrieved from Wikipedia: https://en.wikipedia.org/wiki/2-opt

Wikipedia. (2021, December 1). *A\* search algorithm.* Retrieved from Wikipedia: https://en.wikipedia.org/wiki/A\*\_search\_algorithm

Wikipedia. (2021, December 1). *Singleton pattern*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Singleton\_pattern#Criticism

Wikipedia. (2021, December 1). *Visibility graph.* Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Visibility\_graph