Informatics Large Practical Report

Sarah Bibb

December 2021

1 Software Architecture Description

The application uses 9 distinct classes, and 3 internal classes. The UML diagram displaying each of the classes used in the operation of the drone, can be seen in Figure 1. These classes are as follows:

- **App**: This is the main class of the program, which receives the command line arguments. It creates instances of the classes needed as arguments for later functions, WebServer, Database, and Path, using the necessary information regarding order dates and ports obtained from the arguments. It calls the functions for the orders to be obtained, the path of deliveries to be generated, and for the output information to be written, using the instances it creates.

- **WebServer**: This class contains all of the functionality relating to obtaining information which is stored on the server. It is instantiated in App with the port name provided, and upon construction it creates the string used as the address to connect to the server, and then parses the menu to get the cost of each item, the shop each item is sold in, and the location of each shop, and stores these in variables. This is in case multiple classes need to obtain this information, the server doesn't need to be accessed more times than is necessary.

  It also contains the functions used to parse information on other locations during the runtime of the drone. WhatThreeWords addresses can be parsed, and it can also obtain the information regarding the no-fly zone and landmarks which are in separate files on the server. These are returned as a list of Points and a list of LongLats respectively, for further processing.

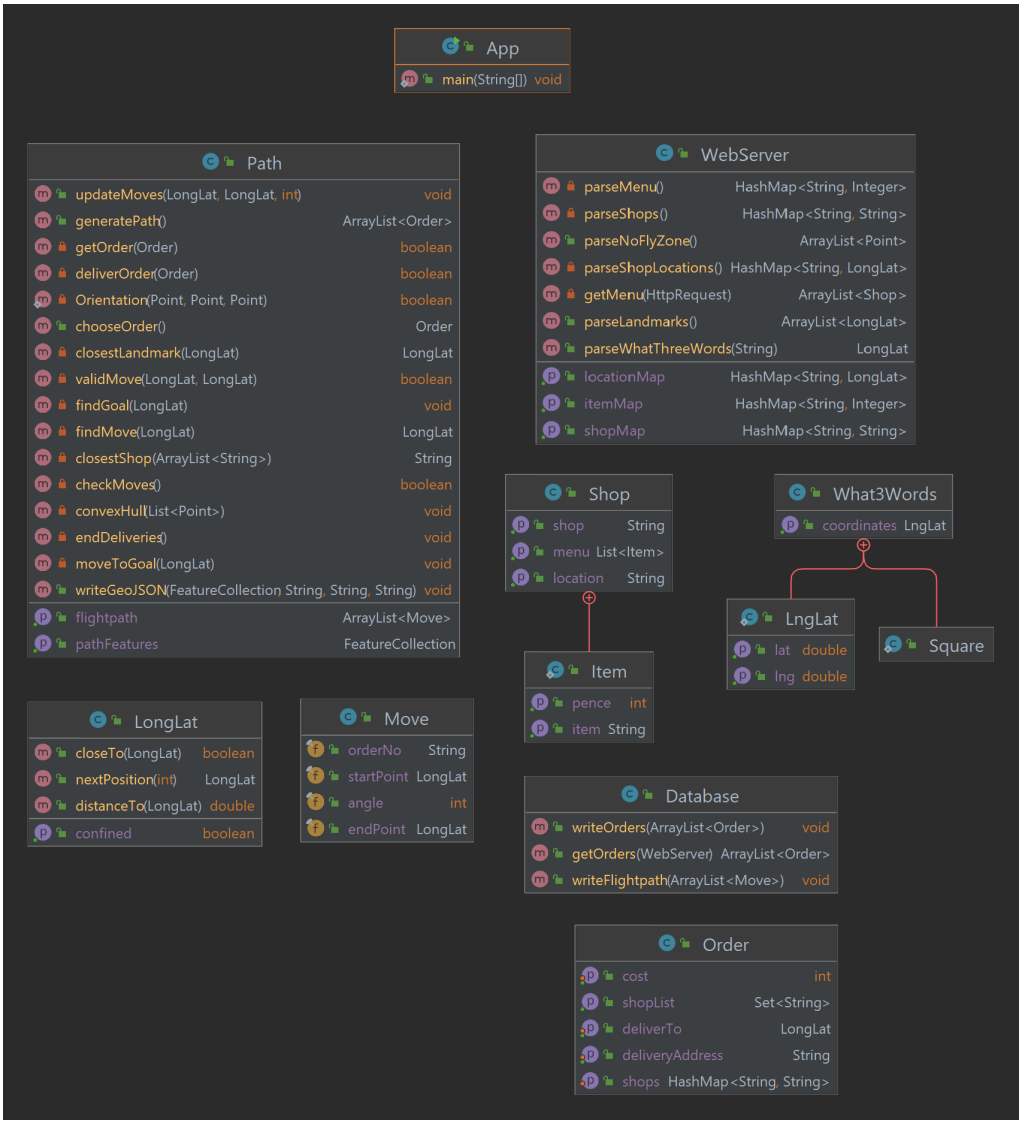

Figure 1: A UML diagram showing the design of each of the classes included in the drone application. LongLat, which represents a point, and Move, which represents a move made by the drone, are grouped with the Path class, which carries out generation of the drone's path, since they are predominantly used there. Likewise, Shop and its internal class Item, and What3Words with its internal classes LngLat and Square, are grouped with the WebServer class, since they are used to translate information obtained from the server. The Order class which represents an order made is grouped with the Database class, where the orders are obtained and the class instances are created.

- **Shop**: This class represents a shop using the information received from the server, which has been parsed from json and assigned to this class which matches the format of the file. It contains the internal class Item, which represents an item sold on a shop's menu. Shop can provide a shop name, WhatThreeWords address, and menu as a list of Items, and each of these Items can provide their name and price.

- **What3Words**: This class represents a WhatThreeWords location. The information is recieved from the server and parsed from json into this class which matches its format. It also contains the internal class Square and LngLat. Square represents the the WhatThreeWords tile, and contains the information on the southwest corner and northeast corner which define the tile. LngLat represents a pair of coordinates, given with a longitude and latitude. The What3Words class can provide the coordinates of a WhatThreeWords tile as a LngLat, and the LngLat class can provide the longitude and latitude of the point it represents.

- **Database**: This class contains all of the functionality relating to obtaining information from the database. It is instantiated in App with the port name provided, and upon construction it creates the string used as the address to connect to the database. It contains the function used to obtain the orders from the database, by obtaining a list of the order numbers of orders which match the date given in the command line arguments, and then creating an instance of the Order class for each one and assigning the details of the order to this instance.

  This class also contains the functions used to write the information on the day's deliveries, namely the flight path and the details of which orders were delivered, to the database once the the drone's path has been successfully generated.

- **Order**: This class represents an order made by a user, which has been obtained from the database. Upon creation the order number is assigned to the instance, and it also contains information on the items in the order, the cost, shops which must be visited to collect the items, and the delivery location. It uses the collection of items and the shops where they are sold from WebServer to find the list of shops.

- **Path**: This class contains all of the functionality used to generate the

path the drone takes while delivering the day's orders. Upon creation it obtains the no-fly zone from the server as a list of points, and uses these to create a convex hull which is used to define the no-fly zone while calculating the drone's moves. It also creates a list of landmarks to use during calculation of the drone's movements.

The method which carries out the main operation of the path generation creates a list of all the orders which are delivered, which can then be used to write to the database the details of these orders. The drone will continue collecting and delivering orders until no more remain, or until it has less than 100 moves remaining, in which case it will return to Appleton Tower to end the day's deliveries.

- **LongLat**: This class represents a point, defined by the longitude and latitude of the point. It also contains functionality for calculating the drone's position, such as checking the distance to another point, confirming if it is close to a chosen point, and finding the new position the drone will be in when it takes a given move.

  An instance of the LongLat class can be created either by providing a pair of doubles, or with an instance of the LngLat class. These classes were chosen to be separate classes because LngLat is defined by its use to parse json files, and as a part of the WhatThreeWords class. It was important for clarity to keep the functionality included in LongLat separate from this, so it has remained as its own class, rather than combining the two.

- **Move**: This class represents a move made by the drone. It is defined by the start and end point of the move, the angle the move was made at, and the order which was being collected or delivered at the time of the move. Null was chosen to be used as the order number when it is returning to Appleton Tower for precision, to ensure it is clear that the deliveries have ended, either because there are no more orders or the drone is low on moves, and that it is ending operations.

2 Drone Control Algorithm

2.1 Order Delivery Description

All of the drone control algorithm is contained in the Path class. As mentioned in the description of this class, before calculation of the drone's journey, the points which define the no-fly zone obtained from the server are used to find the convex hull of these points. This provides a set of lines which defines the outside of the no-fly zone, as it will contain all other points in the list regarding of which building they belong to. This was done to prevent the drone from being stuck in between buildings in the no-fly zone. Due to the fact that the shapes objects in the no-fly zone may be irregular and hard to predict, it was safer to deny the drone entry to the entire area, rather than risk it entering and becoming stuck between buildings, unable to find a path out which doesn't pass through the no-fly zone.

The drone control also relies on a check to ensure the drone doesn't run out of moves before it is able to return to Appleton Tower. After collecting or delivering an order, it will check that the drone has more than 100 moves left. This value was chosen because it was found to be an amount to reliably allow the drone to safely return to Appleton Tower without running out of moves.

The process of calculating the drone's path begins with calculating the total cost of all the orders from that day, so that the percentage monetary value can be calculated at the end of deliveries. It then begins a loop that will continue until either the list of orders is empty, or the move check finds that the drone has 100 or less moves remaining. If either of these are the case, it will begin moving back to Appleton Tower from its current location.

While neither of those are true, the drone will alternate between collecting orders and delivering them. The drone makes use of a greedy approach to select the order which it collects and delivers the user's orders. The next order to be collected is chosen by searching through the current list of remaining orders, and finding whichever one has the highest cost. Once the current order has been chosen, the drone will collect the items. For each shop which needs to be visited to collect items, the drone will check which shop is the closest to its current location, and move towards this shop. Once it is close to the shop, it will hover for a move to collect the items, and the shop is removed from the list which is being searched by the item collection method. It will continue this until the list of shops is empty, meaning that it has

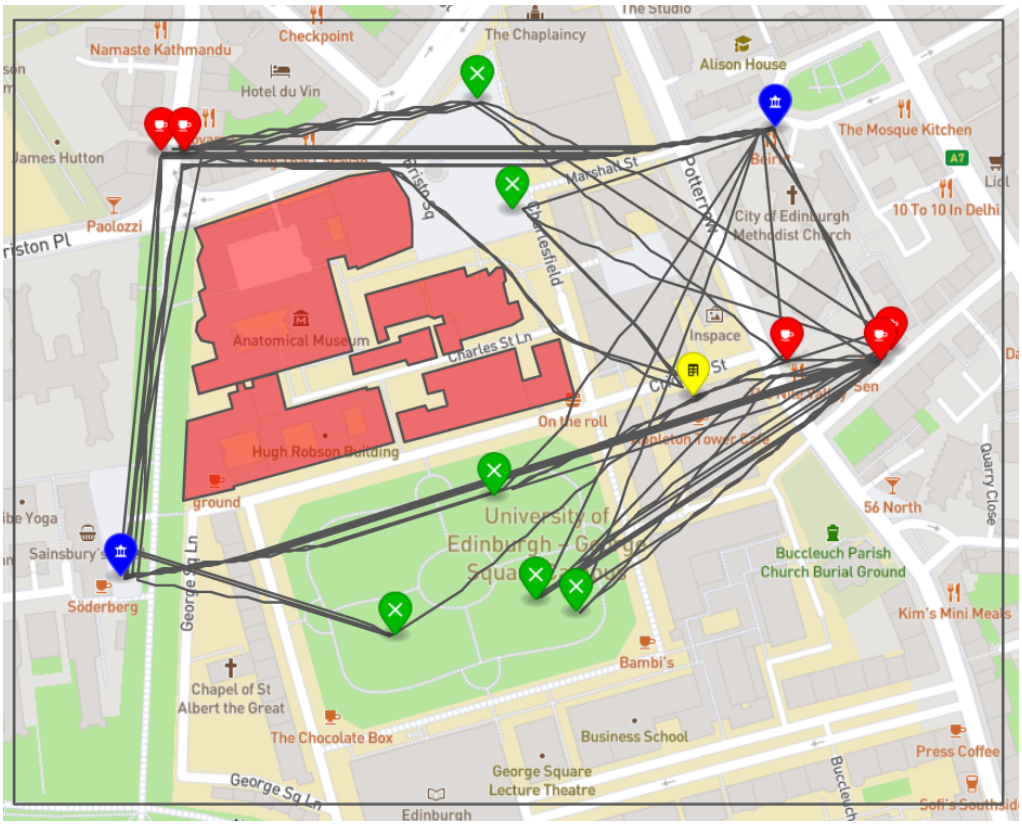

Figure 2: The output GeoJSON file created by the application, showing the drone's operations on 15-03-2023, rendered using http://geojson.io/. The drone was able to deliver 100% of the orders before returning to Appleton Tower.

collected all of the items included in the order.

Once all of the items in the order have been collected, the drone moves towards the delivery location stored in the current order, until it is close to it. Once it reaches the delivery location, it will hover for one move, so that the customer can collect their order. It will then add the cost of the order to the total cost of all orders which have been delivered, so that this can be compared with the total cost of all orders from the day at the end of deliveries. It will also add the order to the list of successfully delivered orders, and remove it from the list of orders which still need to be delivered. This process can be repeated until one of the conditions mentioned above is true and it ends operations for the day.

When the drone ends deliveries, it moves towards Appleton Tower from wherever it currently is, until it is close to it. It then prints the total cost of the orders delivered, the total cost of all orders, the percentage monetary value of the day, the percentage of orders delivered, and the total amount of moves made over the entire day.

2.2 Movement Description

Once the drone's next goal has been decided, be it a shop, delivery point, or Appleton Tower, it will check whether the direct path to the goal would cause it to cross the no-fly zone. If this is the case, it looks for the landmark closest to the current goal which does not require it to cross the no-fly zone, and moves to this instead. Once it is close to the landmark, it will then move to the original goal. This is to assist the drone in finding a more direct path to the goal, and avoid it hugging the edge of the no-fly zone every time it has to reach a goal on the other side. Moving directly to the closest viable landmark and then on to the goal will be more efficient, as moving in a straight line, although potentially taking a slight detour, will take less moves than having to readjust and curve round each line of the no-fly zone.

After the drone has a goal it can reach without crossing the no-fly zone, the application will find the best move from its current location, updating the current location with each new move, until it is close to the goal. The calculation of the moves the drone should take at each point is also conducted in a greedy manner. Taking the current location, it will test the move which would be made at each valid angle between 0 and 360, and finds whichever one gets the drone closest to the goal, provided that this move is also within the confinement zone, and does not cause the drone to cross the no-fly zone.

Whichever move is found will be updated as the new current location of the drone, and the process will be repeated until the drone achieves its goal. Both the list of moves made and the list of lines to be written to the GeoJSON file are updated to reflect each move.

A greedy algorithm was chosen because although it is not confirmed to find the optimal route for the drone, the space complexity is lower than for alternative algorithms such as A*. This algorithm is exhaustive, and requires all nodes to be in the memory to guarantee the optimal path. Considering the amount of points this would result in, greedy was deemed to be preferable, since it still finds an efficient route and will be less taxing for the drone's control system. The use of a greedy algorithm also makes the choice to use the convex hull of the no-fly zone more advantageous. This is due to the fact that the greedy algorithm does not consider anything before or after the current move, and therefore is more likely to get stuck between buildings in the no-fly zone and not be able to find a way out.

2.3 Results of Algorithm

The performance of the drone is judged by the average percentage monetary value it achieves, meaning the percentage of the total cost of the orders which it successfully delivers. A random number generator was used to generate random dates to test the drone, to ensure that the results provided an accurate impression of the drone's performance. 30 dates were used to further improve the accuracy.

Overall the mean accuracy found was 97.91%, which shows that the drone is able to perform well for the vast majority of cases, and it is only when there is a very high amount of orders that it is unable to complete 100% of them. The minimum percentage monetary value found during these tests was 87.67%, which shows that even when the drone is not able to deliver all the orders it still performs well and is able to acquire the majority of the day's income. The GeoJSON output of 2 of the days randomly selected for testing, 15-03-2023, and 25-11-2023, can be seen in Figure 2 and Figure 3.

2.4 Limitations of the Algorithm

Some aspects of the chosen methods for the drone control have limitations which should be considered. Using a convex hull has been very effective for the given case where the no-fly zone is composed of a few buildings of irregular

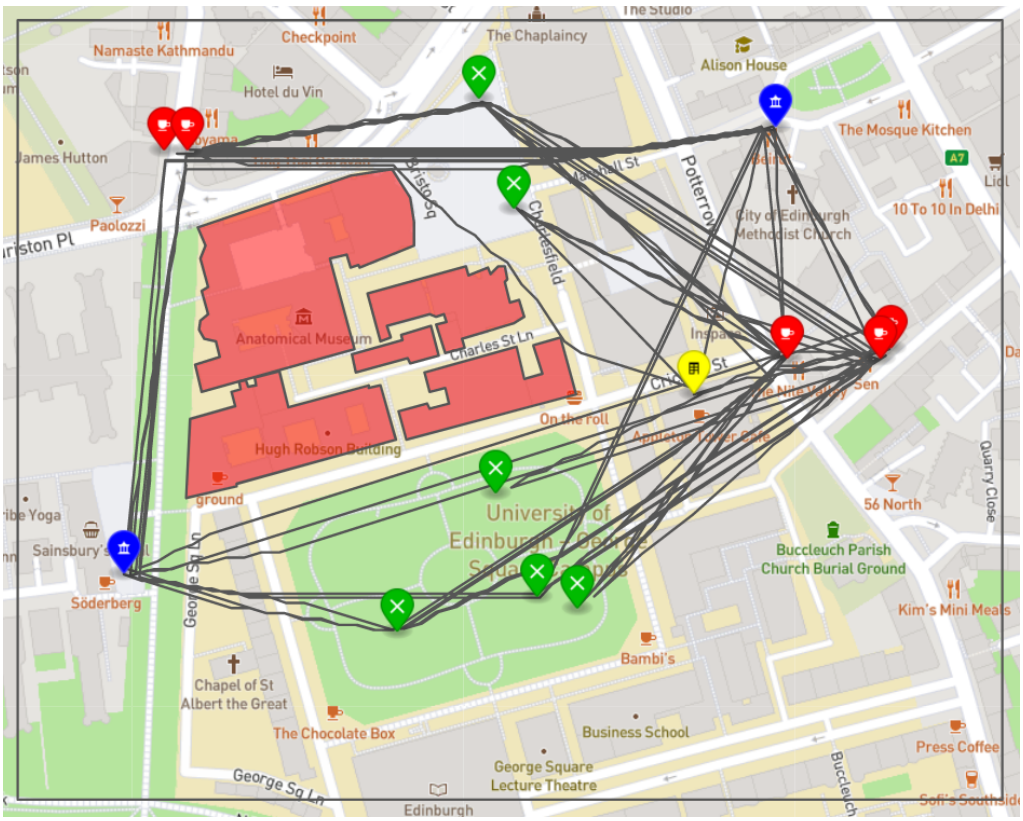

Figure 3: The output GeoJSON file created by the application, showing the drone's operations on 25-11-2023, rendered using http://geojson.io/. The drone was able to achieve 94.34% monetary percentage value on this day, and can be seen collecting an order before the moves check returns that it is low on moves, and it returns to Appleton Tower.

shape which are all located near each other, as it means it can prevent the drone getting caught in between these buildings by simply preventing the drone from entering the area which contains them at all, since it has no reason to. However, there are some cases where this would not be as effective, for example if there was a shop or delivery location located between any of the areas in the no-fly zone. While they cannot be located in the no-fly zone defined on the server, by enlarging the area in which the drone cannot enter, it is made possible that there would be a location that the drone needs to access and cannot. Since the drone control algorithm is specifically to never allow the drone to cross any of the lines which define the convex hull, it would be unable to reach any goal contained in there, and would not be able to complete the current order selected, meaning the process would fail to operate in this situation.

As well as this, if for example part of the no-fly zone is at the top of the confinement zone, and part of it is at the bottom, the resulting convex hull will span the entire confinement zone. This will completely prevent the drone from crossing the area as it should be able to. If it is a particularly unfortunate arrangement may even mean the drone's starting point at Appleton Tower is in the convex hull, meaning it will never be able to find a valid move to leave the starting point. Using the convex hull is efficient in the case presented in this report, but it is important to be aware that it will not be this effective in all cases, and may even be detrimental.

Another limitation of the algorithm used in the application is the method in which it checks the amount of moves remaining, to judge whether it needs to return to Appleton Tower. A value was chosen to allow it always be able to have enough moves to return safely without running out of battery, as it is preferable that the drone be able to return with spare moves remaining, rather than it running out of moves on the way back. However, this system could be improved to plan the next leg of the journey and calculate the amount of moves needed, and use this to confirm whether the drone has enough moves remaining in a more intelligent manner, which may improve the amount of deliveries the drone is able to make.