ILP Coursework 2 Report

Angus Stewart (s1902147)

December 2020

Contents

1 Software Architecture

When reading the coursework specification, it was clear that there were two distinct components to this practical. The first being the **web server** and the second being the **drone**. For this reason, I decided to create two sub-packages within my `aqmaps` package to contain all the classes related to each component. My `App.java` puts the pieces together to form the final application (e.g. by dealing with the command-line inputs and calling the appropriate public methods of objects of the classes defined in the sub-packages).

The overall structure is shown in Figure 1. The links between the classes indicate a direct relationship (e.g an object of one class is an attribute of another, or a method of one class utilises an object of another).

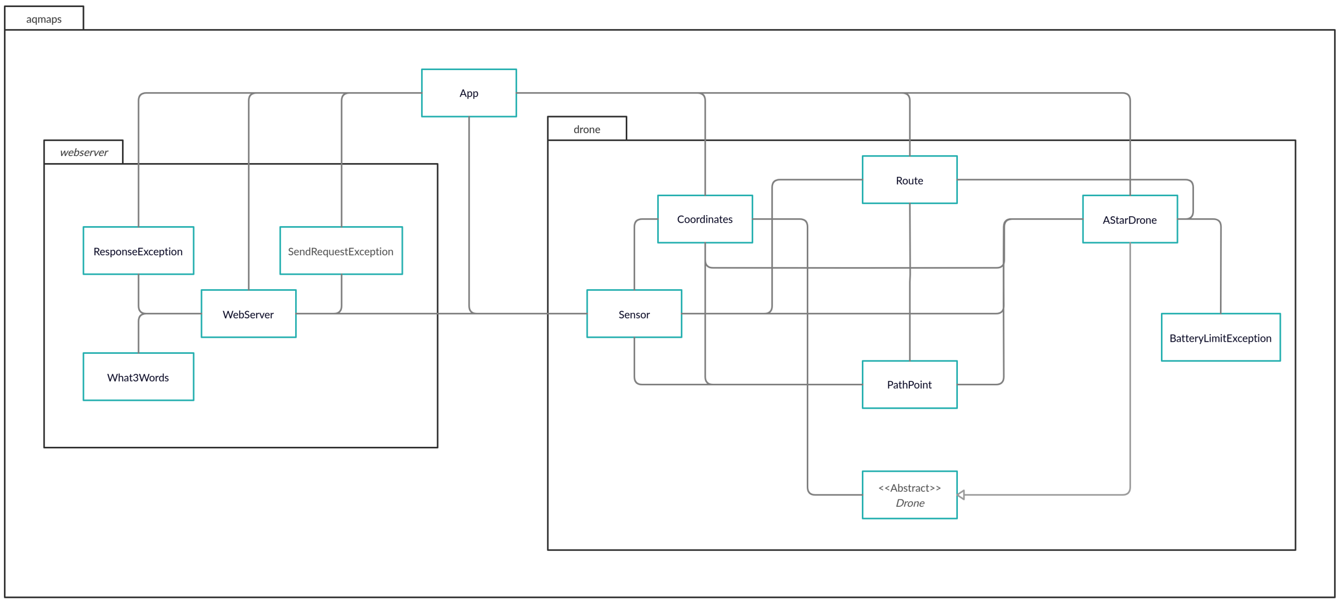

Figure 1: Class diagram of `aqmaps`

1.1 `webserver`

The web server sub-package deals exclusively with **obtaining information from the web server**. Because this is a very important part of the practical, it felt appropriate to dedicate an entire sub-package to this.

The sub-package contains 4 classes:

1. `WebServer`
2. `What3Words`
3. `SendRequestException`
4. `ResponseException`

The `WebServer` is the main class in this package and provides a layer of abstraction over all communication to the web server. It made sense to create a web server class because the web server is an standalone entity within our system. The `WebServer` class models this and provides some useful methods for retrieving data from the server. The other three classes support the main `WebServer` class. The `What3Words` class is used to store the JSON information obtained from the `words/` directory on the web server as a Java object. This is required by `gson`. We also use the `Sensor` object from the `drone` sub-package when obtaining sensor information from the web server. The reason that class was included in the `drone` sub-package and not this package was because it did not make much sense to store a class that represents a physical sensor in a package dedicated to dealing with a web server. It made more sense to include it in the drone package where the drone is modelled due to the close relationship that these two objects have in this practical.

The other two classes (`SendRequestException` and `ResponseException`) were created to abstract over the specific exceptions that can occur when communicating with the web server. The `SendRequestException` abstracts over the exceptions related to the creation and sending of the HTTP request (e.g. if an invalid URI was provided). The `ResponseException` abstracts over the exceptions relating to the HTTP reponse from the web server (e.g. if a 404 error was returned). This provides a cleaner interface for the user of the web server that hides away unnecessary complexity.

1.2 `drone`

This sub-package consists of a set of classes related to the drone's **internal representation of the world**. In other words, these classes together model the problem detailed in the specification of determining and presenting the flight path of a drone around a geographical area with no-fly-zones to collect readings from a set of sensors. This may seem like a lot of disparate ideas, but they are in fact very closely related.

There are 7 classes in this sub-package:

1. `Drone`
2. `AStarDrone`
3. `Sensor`
4. `Route`
5. `PathPoint`
6. `Coordinates`
7. `BatteryLimitException`

The `Drone` class models the **physical properties that affect the drone**. For example, it defines the method `move` which models how to drone moves in the physical world. `getDistance` represents the measure of distance between two points in the real world. It also has attributes that model the confinement area that constrain the drone as detailed in the specification and provides methods for modelling the no-fly-zones. All together this class models the relationship between the drone and the physical world as detailed in the specification. For this reason, the `Drone` class is abstract as instantiating it on its own is undefined. Instead this class is inherited by all drones to allow it to obey the physical contraints posed by the specification.

This brings us to the `AStarDrone` class which models the **behaviour of the drone**. It inherits from the `Drone` class to understand the physical world, then it uses its own methods to navigated within that world. This is where the drone control algorithm (detailed in section 2) is implemented. The class retrieves the initial starting conditions of the world (i.e starting position, sensors, and no-fly-zones) and calculates an appropriate route to visit all the sensors and return to the starting position.

In my original implementation, I only had one `GreedyDrone` class which contained the implementation of both `Drone` and `AStarDrone`. I then decided to create a new drone control algorithm based on the A* search algorithm as the greedy approach I was using was giving me some problems. It was when I was doing this that I noticed that some functions and attributes were being duplicated in both the `GreedyDrone` class and the then newly created `AStarDrone` class. This was the original motivation behind creating an abstract class that 'pulled out' this shared implementation. Upon closer inspection I realized that the code I was pulling out reflected the rules of the physical world in which the drone was operating under as described in specification (e.g distance of a single move of the drone, the distance measure, the representation of the confinement area, etc). This was a pleasant surprise.

The `Route` object is the third main class within this sub-package. This class arose due to a strong emphasis that the specification placed on the route of the drone as a GeoJSON map and as a flightpath in txt format. I felt that modelling the route itself was appropriate as the GeoJSON map and the flightpath were simply different representations of the same underlying 'object' i.e the route. This class provides convenient methods to obtain both the flightpath and the GeoJSON map of the route as `.txt` and `.geojson` files respectively.

The two classes `Coordinates` and `PathPoint` both represent important concepts that map to the real world. The `Coordinates` class represents the latitude and longitude values of a position on the map as described in the specification. This abstraction is incredibly useful for representing the position of objects such as the drone or the sensors. The `PathPoint` object represents a single movement of the drone from one position to another. This class was created to represent the concept outlined in the specification for each line of the flightpath text file. It is also used within the drone to represent the global route as a sequence of drone movements and the drone control algorithm as a way to represent the different possible movement options that the drone has at each particular step.

Finally, the `BatteryLimitException` class is used to represent the event when a route calculated by the drone control algorithm is longer than the battery of the drone allows. By raising this exception the control algorithm can react and create a contingency plan (e.g by returning to the starting position). This class is not essential but proved to be convenient when implementing the control algorithm.

2 Drone Control Algorithm

The specification document describes a modified version of the well known travelling salesman problem (TSP) where we look to visit each sensor and return to the starting position. Of course, it is not exactly the same problem as the path to each sensor is not a single link but rather a series of restricted movements. Hence, the combination of the travelling salesman problem being NP-complete and the movements of the drone made the complexity of the problem very apparent from the beginning.

Knowing that finding an algorithm that would solve the entire path planning problem optimally would not be feasible, I broke the problem into two sub-problems. The first is the order in which the drone visits the sensors and the second is finding the sequence of valid moves to get from one coordinate position to another.

With these two components, we can find the entire route:

1. Get the next **unvisited** sensor given the current position of the drone
2. Find the route to the sensor
   (a) If enough battery, move the drone along the route and mark the sensor as visited
   (b) Otherwise, skip to step 4
3. Go to step 1 until all sensors have been visited
4. Find the route back to the start given the current position. If not enough battery to complete the entire route, follow the route until we run out of battery to return as close to the start as possible.

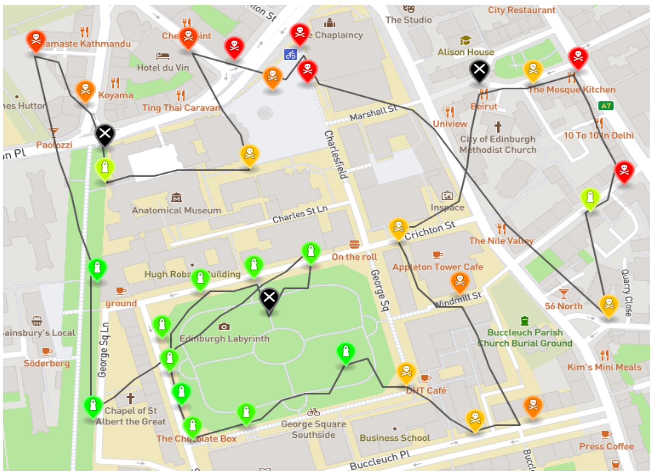

Figure 2: 25th of October 2021

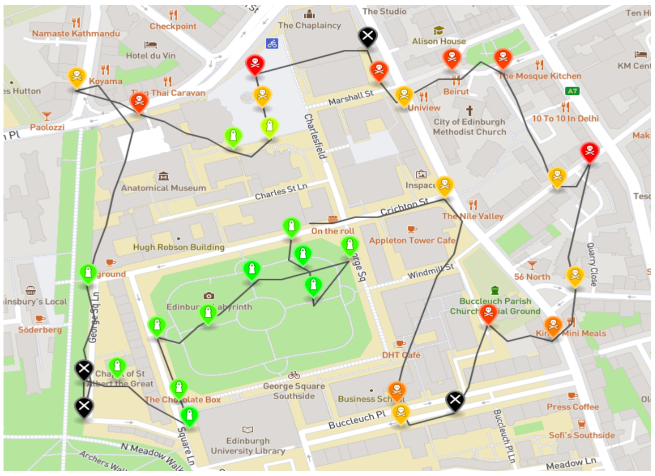

Figure 3: 26th of December 2021

2.1 Ordering of the Sensors

The first problem is determining the order in which the drone should visit the sensors. To keep things simple, I chose a greedy approach, and this is what the drone uses in the implementation. The algorithm calculates the straight-line distance between a given coordinate position (e.g. current position of the drone) and the coordinates of a list of sensors (e.g. list of unvisited sensors) and returns the closest sensor.

Although this algorithm is simple and runs very quickly (linear in the number of sensors), it is unlikely to produce an optimal sequence of sensors to visit. We can see this with Figure 2 where the drone has to take long routes across the map to reach sensors that it missed. In addition, it does not take into account the no-fly-zones of the map and how this affects the distances between sensors. But given the TSP nature of this problem, obtaining good solutions requires significant effort in implementing complex algorithms. Hence, I believe this greedy algorithm is a good starting point due to its balance of simplicity and performance.

It is worth noting that the drone only looks for the next sensor and does not pre-compute the order of all sensors. This decision was taken because the drone does not necessarily reach the exact coordinates of the target sensor and hence pre-computing everything from the beginning would be inaccurate.

2.2 Routing to the Next Position

Given the current position of the drone and the target position (e.g coordinates of the next sensor or of the starting position), we must calculate a valid route between the two points. Note that we do not need to end up **at** the target position, just within a certain distance from it (e.g 0.0002 for sensors).

This can be thought of as a tree search problem. The tree represents all possible sequences of moves (represented by the edges of the tree) that the drone is allowed to make, and the nodes represents coordinate positions with the root being the current position of the drone. Note that the branching factor for this tree is 36 (though it may be less if the movement of the drone is restricted e.g due to being near no-fly-zones).

In my implementation, I use a modified A* search algorithm to find the path down the tree to a node that is within range of our target position. The algorithm consists of two lists: `closedPathPoints`, which stores all the nodes that we have visited (expanded) and `openPathPoints`, which stores all the nodes that we have yet to visit but whose parents have been visited.

1. We begin with only the root node in `openPathPoints`. Remove this node from the list and 'expand' it to generate all **valid** children nodes [1]. These child nodes represent positions the drone can move to in one step that does not violate any of the rules (e.g regarding no-fly-zones).
   (a) Each child node gets assigned a `distanceScore` (among other attributes such as `prev`) which is the sum of the travelled distance from the root to the current node and the straight-line distance from the current node to the target position.
   (b) We then select 5 of the child nodes with the smallest `distanceScore`. We do this to reduce the time complexity of the algorithm as using all child nodes would cause the algorithm to run slowly.
2. Add the 5 nodes to `openPathPoints` and add the root to `closedPathPoints`
3. Select the node from `openPathPoints` with the lowest `distanceScore`. If in range of the target position, skip to step 6, otherwise expand it as per step 1.
4. Before adding each of the 5 nodes to `openPathPoints`, we do two checks:
   (a) If their coordinates match with a node already in `openPathPoints`, we look to see if it has a smaller `distanceScore`. If it does, we add the new node and remove the old one. Otherwise, we discard the new node.
   (b) If the node was not added to `openPathPoints` or discarded in part (a), check to see if the coordinates match with a node already in `closedPathPoints`. If it does, discard the node.
5. Add the expanded node (from step 3) to `closedPathPoints` and go to step 3

[1]To check if a child node is valid, we take the line segment formed by the parent node and the child node and see if it intersects (using the `java.awt.geom`) with an edge of the drone confinement area or no-fly-zones. If it does, it is invalid.

6. Reconstruct the route from the final node from step 3 by following `prev` (which points to the node's parent) until we reach the root.

Notice that this algorithm will always produce a route of at least length 1 because it always expands the root node before checking if any of the nodes are within range of the target position. This is required as per the specification. However, because we do not need to move before finishing, when calculating the route back to the start we check before step 1 whether the root is within range of the target position.

This algorithm has the advantage that it can handle no-fly-zones without much modification (we just don't generate the child nodes that cross the no-fly-zones). This contrasts with my original algorithm that used a greedy approach which would get 'stuck' oscillating between two nodes if the target position was on the other side of a no-fly-zone. This is because unlike the A* search algorithm, the greedy algorithm did not take into account the distance from the root node. However, the time complexity of the A* algorithm is not as good (hence step 1.b) so the time the algorithm takes to find routes over larger distances become much more noticeable which puts the scalability of this algorithm into question.

It is also worth mentioning here, that although **the algorithm is the same**, in the actual implementation we use `PathPoint` objects, which represent the movement of the drone and hence would correspond to the edges of the tree, rather than nodes as described above. However, `PathPoint` stores the nodes in `endPos` (and its parent in `startPos`) and attributes such as `distanceScore` correspond to the position of `endPos` and so the implementation details of the algorithm is very similar.

3 Class Documentation `aqmaps`

3.1 `webserver`

3.2 `drone`

4 References

The following sources were used either to provide inspiration for my implementation or to directly aid in it (e.g. through example code or pseudocode):

1. Wikipedia - Motion planning
2. Wikipedia - Any-angle path planning
3. Wikipedia - A* search algorithm
4. Wikipedia - Best-first search
5. Stackoverflow - How to compare two java objects

The following sources were used to help with the report either for the figures or for the formatting:

1. geojson.io
2. Creately
3. Wikipedia - Class diagram
4. Tex StackExchange - LaTeX figures side by side
5. Tex StackExchange - Remove ugly borders around clickable cross-references and hyperlinks