ILP Coursework 2 Report

Angus Stewart (s1902147)

December 2020

Contents

1 Software Architecture

2 Drone Control Algorithm

The specification document describes a modified version of the well known travelling salesman problem 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 would be not be feasible, I broke the problem down into two sub-problems. The first is the order in which the drone is to visit 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 route:

1. Get the next sensor given the current position of the drone
2. Find the route to the next sensor
   (a) If enough battery, move the drone along the route and connect to sensor
   (b) Otherwise, skip to step 4
3. Go to step 1 until all sensors have been visited
4. Find route to the starting position given the current position. If we do not have 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.

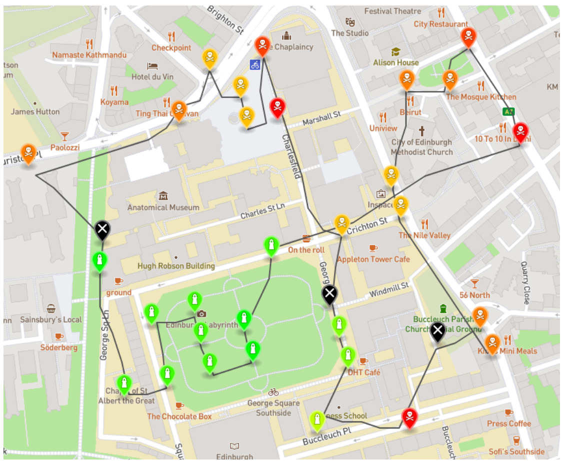

Figure 1: Geo-JSON map for the 25th of December 2021

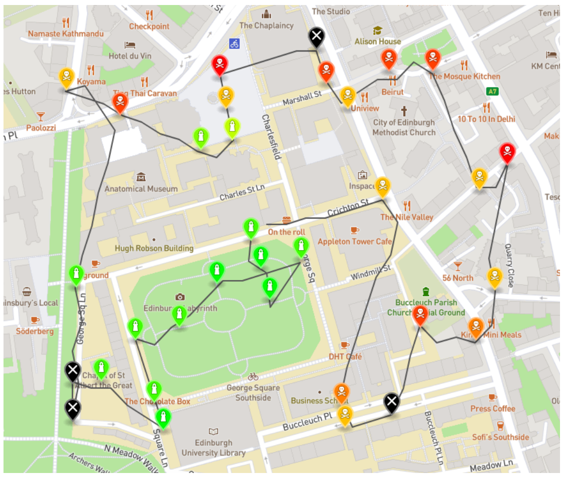

Figure 2: Geo-JSON map for the 26th of December 2021

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 both conceptually and technically, I decided to use a greedy approach, and this is what the drone uses in the implementation. The drone calculates the straight-line distance between its current coordinates and the coordinates of all **unvisited** sensors and selects the sensor that is closest (the distance measure is the same as what is mentioned in the specification i.e. Pythagorean distance).

It is worth noting that the drone only looks for the next sensor and does not pre-compute the order of all sensors. Once it has reached the next sensor, it will recompute based on its coordinates and the coordinates of all remaining unvisited 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 (this could be the coordinates of the next sensor or the coordinates to the starting position if we have visited all the sensors), 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 (0.0002 for sensors and 0.0003 for the starting position).

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, with the nodes representing coordinate positions (and 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.

1. We begin with only the root node in `openPathPoints`. Remove this node from the list and 'expand' it to generate all its children nodes. These children nodes represent **valid** positions one step away from the current node.
   (a) Each child node gets assigned a `distanceScore` 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.
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 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, check to see if the coordinates match with a node already in `closedPathPoints`. If it does, discard the node.
5. Add the current node (from step 3) to `closedPathPoints` and go to step 3
6. Reconstruct the route from the final node from step 3 by following `prev` (which points to the previous node in the path) 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. If it is, then the above algorithm is not run.

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 `prev` or `distanceScore` correspond to the position of `endPos` and so the implementation details of the algorithm is actually not that different from above.

3 Class Documentation

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