Informatics Large Practical Coursework 2 Report

Harry Lennox

S1833978

Software Architecture Description

Introduction to Classes

The program consists of 9 classes total. These are:

- App
- FlightController
- GeoJSONHandler
- ServerHandler
- FlightPath
- NoFlyZone
- Sensor
- Move
- Coords

The App class is the main class of the program, where all other function calls originate from. The FlightController, GeoJSONHandler and ServerHandler are the three major classes of the program and rely upon the other classes such as FlightPath in order to function.

Breakdown of the Problem

The task of creating the program can be broken down into several separate issues. Firstly, there is the issue of retrieving data from the web server and parsing that data into useful classes. Second, there is the issue of using this parsed data to create valid GeoJSON which can then be rendered into the desired map. Finally, there is the issue of the pathfinding algorithm itself, and representing the various unique factors that need to be accounted for in order for it to function correctly. We also require representation of the problem space itself, with classes for features of the map and drone.

Identification of Classes

The three 'major classes' described above were chosen to represent each of these three major issues, and each class handles everything to do with that section of the task. ServerHandler was chosen to take care of all functionality relating to retrieving and parsing data from the web server, GeoJSONHandler was chosen to handle all major GeoJSON operations and FlightController was chosen as the main class for the drone's flight algorithm. These classes offer encapsulation, so that the base App class does not contain any excessive import statements or functionality.

The space in which we are operating is a 2d map with some restricted areas as well as various sensors at locations around the map, and a drone that can fly fixed distances in a given compass direction. To this extent, several classes were chosen to represent key attributes of this workspace. Coords, Sensor and NoFlyZone were all chosen to represent the important attributes of the map – where on the map we are, the state of all sensors on the map, and the boundaries of areas we cannot enter. Similarly, Move and FlightPath were chosen to represent the movements of the drone. Move represents a single move, whilst FlightPath contains the implementation for stringing together moves between a given start and end point.

The architecture of the program is fairly simple, which is intentional. Since our problem space can be fully expressed in only 9 classes, to break them down further or implement things such as subclasses would just introduce needless complexity. The design can respond to changes in the specification, even without these features. Constant values can be represented as fixed variables inside their specific class, which can be changed easily. If a different server were to be used, it would just require changes to the ServerHandler – in fact the ServerHandler could be thrown away completely and replaced with anything else that implements the same API. You could create subclasses for something such as FlightController, changing the algorithm with each one, but the current algorithm is more than sufficient for the task. The simplicity and compactness of the program can be considered a benefit. In the next section, we will provide detailed documentation for each of these 9 classes.

Class Documentation

App

App contains one main function which executes the program as well as a private function outputFlightPath which creates the flightpath text file. The class itself contains no functionality other than outputFlightPath, and simply uses instances of the classes defined below, so we will not cover it in great detail.

In the following sections we will briefly document the fields and methods of each class. Any getters or setters will be detailed below, where there is no comment there are none.

FlightController

Fields		
Type	*Name*	*Description*
List<Coords>	coordsList	Passed into the constructor, this is a list of the positions of the sensors as Coords objects.
Coords	START_POINT	The start point of the drone's flight, passed in the constructor.
List<String>	sensorNames	The w3w names of all the sensors, passed in the constructor. Used for generating Moves.
List<NoFlyZone>	noFlyZones	A list of NoFlyZones that define the areas the drone may not enter, passed in the constructor.
FlightPath[][]	pathMatrix	Generated from the coordsList, this is a 2D array containing FlightPath objects from each of the given sensors to every other sensor and is used by the algorithm to gauge distance.
int[][]	connectivityMatrix	Represents the connections made between sensors in the order we generate, where 0 is no connection and 1 is a connection to that sensor.

Methods		
Type	*Name*	*Description*
List<FlightPath>	generateOrder()	This is the main function for the drone's pathfinding algorithm. It is an implementation of the sorted edges algorithm discussed in length in the next section, and handles calculating the order in which to visit each sensor.
List<FlightPath>	createPathList(List<Integer> order)	Takes the ordered list of sensors given by generateOrder and creates

		the final list of FlightPath. Importantly, it recalculates the actual FlightPaths to ensure the next begins where the previous finished.
List<Integer>	traverseConnections(int from)	Uses the connectivityMatrix to follow the line segment from the point *from* and uses this to form an integer order of sensors.
FlightPath[][]	generatePathMatrix(List<Coords> coords)	Creates FlightPath objects between all sensors, allowing us to use the number of moves as an estimate of distance between any two sensors.
Boolean	isDegree(int node, int degree)	Uses the connectivityMatrix to output the degree of a given sensor, or in other words, how many other sensors it is connected to.
Boolean	wouldCompleteLoopEarly(int from, int to)	Checks whether connecting the two given sensors would complete the loop before every other sensor has been visited by using traverseConnections and testing whether the points *from* and *to* are connected to each other already through other sensors.
Triplet<Integer, Integer, Integer>	getShortestValidDist(int index, List<Integer> fullNodes)	Finds the closest unvisited sensor to the given one, ensuring no conditions of the algorithm are broken in the process.

GeoJSONHandler

Fields		
Type	*Name*	*Description*
Coords	START_POINT	The start point of the drone's flight, passed in the constructor.

Methods		
Type	*Name*	*Description*
Feature	generateMarker(Sensor loc, Coords coords)	Takes a location and assigns it a marker with colour and symbol based on the specifications. It returns the marker as a GeoJSON Feature.

Void	outputJSON(List<Feature> featureList, String day, String month, String year)	Takes the finished path as a list of Features and writes it to a GeoJSON file, naming it according to the day of the flight.
List<NoFlyZone>	parseNoFlyZones(String data)	Takes the String data returned from the web server as JSON and parses it into a list of Features, using each to construct a NoFlyZone object. It then returns the final list of NoFlyZones.
Feature	generatePath(List<FlightPath> pathList)	Takes in the list of FlightPaths calculated by the flight algorithm, and constructs a GeoJSON LineString out of each individual point the drone flies to in sequence.
String	getSymbolFromReading(float reading)	Helper function decides what icon a marker should have based on its reading.
String	getRGBFromReading(float reading)	Helper function which decides what colour to use for a given marker based on its reading.
Boolean	inRange(float val, int min, int max)	Helper function that stands in for min <= val < max

ServerHandler

Fields		
Type	*Name*	*Description*
String	uri	The location of the web server given the port passed in through the constructor.

Methods		
Type	*Name*	*Description*
Sensor[]	getSensorData(String year, String month, String day)	Fetches the air quality data for the specified day from the web server and parses it into Sensor objects using Gson.
Coords	covertWordToCoords(String location)	Converts a given w3w word into its corresponding coordinates by fetching its details.json file from the web server. It then parses this information into a Coords object.
String	getNoFlyZoneData()	Fetches the GeoJSON data for the NoFlyZones from the web server,

		returning it as a String to be parsed later.
String	getServerData(URI fullUri)	Performs the actual HTTP requests for the above functions, constructing a request using the Java HttpClient and sending it to the web server, then returning whatever it finds.
URI	buildUri(String directory, String[] pathArgs, String file)	Helper function that builds a full uri to access a server resouce. Directory is the base directory of the web server we are interested in, pathArgs is an array containing the in-between steps required to get to the final directory, and file is the name of the file in that directory we are interested in.

FlightPath

Fields		
Type	*Name*	*Description*
Double	MOVE	The constant value of a single move. It is final and always set to 0.0003.
Int	moveCount	The number of moves this particular FlightPath takes to complete. This is the size of the moveList.
List<Move>	moveList	A list used to store each Move made by the algorithm along its path.
Coords	startPos	The position on the map the drone starts at, passed in through the constructor.
Coords	endPos	The position on the map the drone is aiming to get to, passed in through the constructor.
List<NoFlyZone>	noFlyZones	A list of NoFlyZone objects representing the areas of the map the drone cannot enter.
String	sensorName	The name of the Sensor object the drone is trying to get to, needed to attach to the final Move that gets it there for output.

Methods		
Type	*Name*	*Description*
List<Move>	calculateFlightPath()	The main function of the class which performs the FlightPath algorithm detailed in the next section and returns a list of the

		Moves it took to be used as the moveList.
Boolean	leavesConfinementZone(Coords newPos)	Helper function that checks whether the current position of the drone is beyond any of its limits.
Int	calculateNewAngle(Coords currPos, int optimalAngle, int previous)	The correction algorithm detailed in the next section. Used to find the best move to make when the optimal one enters a NoFlyZone.
Int	angleDistance(int angle1, int angle2)	Helper function which finds the distance between any two angles in terms of degrees.
Boolean	entersNoFlyZone(Coords currPos, Coords newPos)	Helper function that utilises NoFlyZone.intersects() and checks each to ensure that the current move would not enter any no-fly zone in the noFlyZones list.
Coords	getNewLocation(Coords currPos, int roundedAngle)	Performs one move of the drone, using trigonometry to calculate where its new location is when it travels at the given angle.
Double	getAngleBetween(Coords c1, Coords c2)	Uses the atan2 function to calculate the angle between any two Coords objects, used to find the optimal angle to travel at.
Boolean	isClose(Coords c1, Coords c2)	Helper function which checks whether the drone is in range of a sensor. Returns true if the Euclidean distance between the two Coords is <= 0.0002.
Double	getDistBetween(Coords c1, Coords c2)	Helper function which calculates the Euclidean distance between two Coords objects.

There are getters for the moveList and moveCount, but not for any other fields.

NoFlyZone

Fields		
Type	*Name*	*Description*
List<Coords>	pointList	The list of points that make up the polygon of the NoFlyZone.
String	name	The name of the NoFlyZone – e.g. Appleton Tower.

Methods		
Type	*Name*	*Description*
List<Coords>	convertCoords(List<List<Point>> points)	Converts from the given GeoJSON Polygon's coordinates (given by the API as a List<List<Point>>) to a list of Coords, which is assigned to pointList.
boolean	intersects(double x1, double y1, double x2, double y2)	Goes through each point in the pointList, using Line2D's linesIntersect function to check whether the given line segment enters the NoFlyZone. It takes one point in the pointList and the point after it and checks whether that line and the one defined by the input parameters intersects.

The name field has a getter, but not a setter, as it is final.

Sensor

Fields		
Type	*Name*	*Description*
String	location	The w3w location of the given Sensor, which can be used to find its coordinates.
Double	battery	The battery level of the Sensor represented as a double. If below 10%, the reading may be unreliable.
String	reading	The air quality reading of the Sensor, represented as a String as low battery sensors may have readings such as "NaN" or "null".

All three fields have getters, but not setters as they are final values.

Move

Fields		
Type	*Name*	*Description*
Coords	start	The start point of the Move, represented as a Coords object.
Coords	end	The end point of the Move, represented as a Coords object.
Int	bearing	The bearing at which the drone travels. It is an integer value between 0 and 350.

String	sensor	The name of any sensor this Move brings the drone close to. For most moves this will be null, but when it is not it will be a w3w location.

All four fields have getters, but not setters as they are final values.

Coords

Fields		
Type	*Name*	*Description*
Double	lng	The longitude value of the coordinates.
Double	lat	The latitude value of the coordinates.

Both fields have getters, but not setters as they are final values.

Drone Control Algorithm

The Pathfinding Problem

There are several factors which complicate the pathfinding algorithm of the drone, however when boiled down to a basic level, the control algorithm needs to be able to generate a Hamiltonian Circuit between 33 vertices (sensors) and a starting point. Finding the optimal circuit is a complicated and intensive task, but we are not required to find the optimal circuit, just one which will complete in 150 moves or less. This restriction means we cannot rely on a greedy algorithm such as Nearest Neighbour. However, our algorithm also needs to be reasonably fast and efficient, so we cannot rely on a Brute Force Algorithm. Our graph is complete, meaning the total number of routes (removing duplicates) is calculated as (n-1)!/2 where n is the number of nodes. Including the starting point, we have 34 places to visit, so there are ~4.34×10^{36} different routes. We need to find a middle ground between simplicity and optimality. The algorithm that was chosen is the Sorted Edges algorithm.

The Sorted Edges Algorithm

The Sorted Edges algorithm essentially works as follows:

1. Sort the edges of the graph by their lengths in ascending order.
2. Choose the edge of shortest length and use it in the circuit, unless either of the following conditions are met:
    a. The circuit is completed before all vertices have been included.
    b. Choosing this edge would create a vertex of degree 3.
3. Repeat until a Hamiltonian Circuit is formed.

This algorithm ensures we always take the shortest paths first and builds a path without the restriction of doing so in an order. It does not ensure the optimal path, but as previously stated we only need a reasonable degree of optimality for the task. It is also reasonably efficient, being more complex than Nearest Neighbour, but far less complex than the brute-force method. It is for these reasons it was chosen.

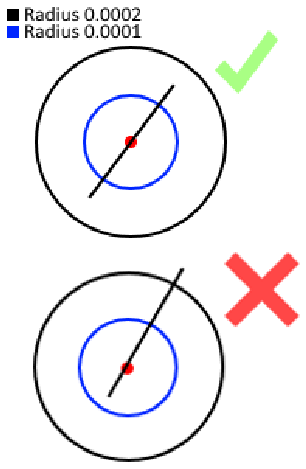
The Complications

The sorted edges algorithm is designed to take a single straight line distance between any two nodes, but in reality our sensors are connected by a drone flight path consisting of some number of 0.0003 degree moves, with bearings restricted to multiples of 10. There are also some areas in which the drone cannot enter, the no-fly zones. For these reasons, simple Euclidean distance would not work. Instead, the FlightPath class was created and the pathMatrix was used to represent the distances between any two points based on these flightpaths. The chosen distance measure was now the number of moves the drone must make to get between the two sensors, which is a measure that takes into account avoiding the no-fly zones. Generating a path therefore comes down to two factors – choosing the right order to visit each sensor and creating the correct flightpath between every connected sensor. Ensuring both of these are reasonable is the goal of the overall pathfinding algorithm. The Sorted Edges method discussed in the previous section handles the former, whilst the latter is handled by the FlightPath class itself, something which warrants further discussion.

The FlightPath Algorithm

A vital part of the pathfinding algorithm is its ability to make the correct moves at the level of the drone. The FlightPath class was created to form these strings of moves. The overall pseudocode for generating these moves is shown below:

```
1. Until the current position of the drone is close to the desired end point…
    a. Find the optimal angle between the current position and the end point.
    b. Round this to the nearest multiple of 10 and calculate where the drone would
       end up if it took this move.
    c. If this point is not outside the confinement zone, and not inside a no-fly zone
       proceed to step f. Otherwise, continue.
    d. Calculate the best new angle to travel at which keeps us within the allowed
       area using the correction algorithm.
    e. Calculate the new position of the drone.
    f. Add this move to the list of moves and set the drone's current position to its
       new position.
2. If our flightpath is of length zero, move once in the optimal direction (or as close to it
   as is valid). Flightpaths must be of length >0.
3. If performing this move takes us out of range, move back to
   the previous location that was in range.
```



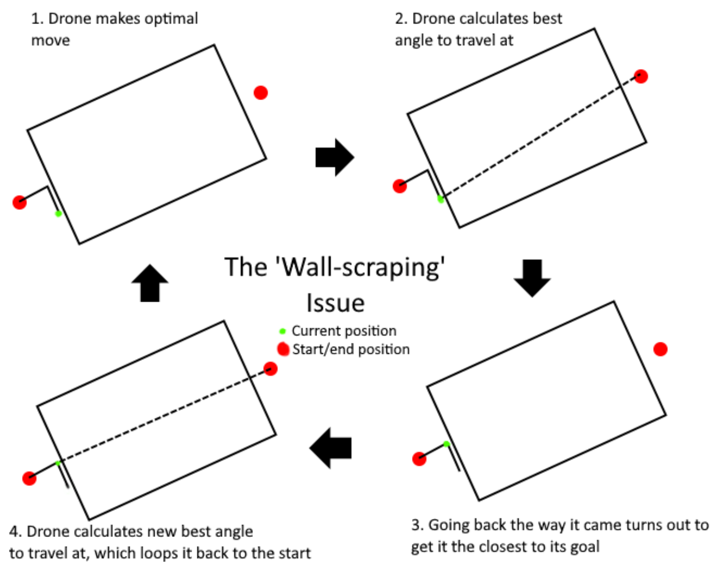
We have a restriction that all flightpaths must be at least one move, detailed in steps 2 and 3. If a flightpath of length 0 was allowed, it would break the restriction that the drone's lifecycle consists of making a move and *then* scanning any nearby sensor. By moving once in the optimal direction, we ensure that the drone will remain in range unless it was previously < 0.0001 degrees from the sensor. The diagram opposite illustrates this scenario. In the unfortunate event that this happens, we simply use a move to return to that previous location. This ensures we remain in range, and never have a path with less than one move.

The correction algorithm is as follows:

```
1. Increment the optimal angle until we find an angle at which we remain inside the
   allowed area.
2. Decrement the optimal angle until we find an angle at which we remain inside the
   allowed area.
3. Clean these outputs to remain in the range of 0-350 degrees.
4. If there has been a previous move, go to step 5, else compare the incremented and
   decremented angles and return the one closest to the optimal angle.
```

5. Compare the incremented and decremented angles and return the one closest to the angle of our previous move.

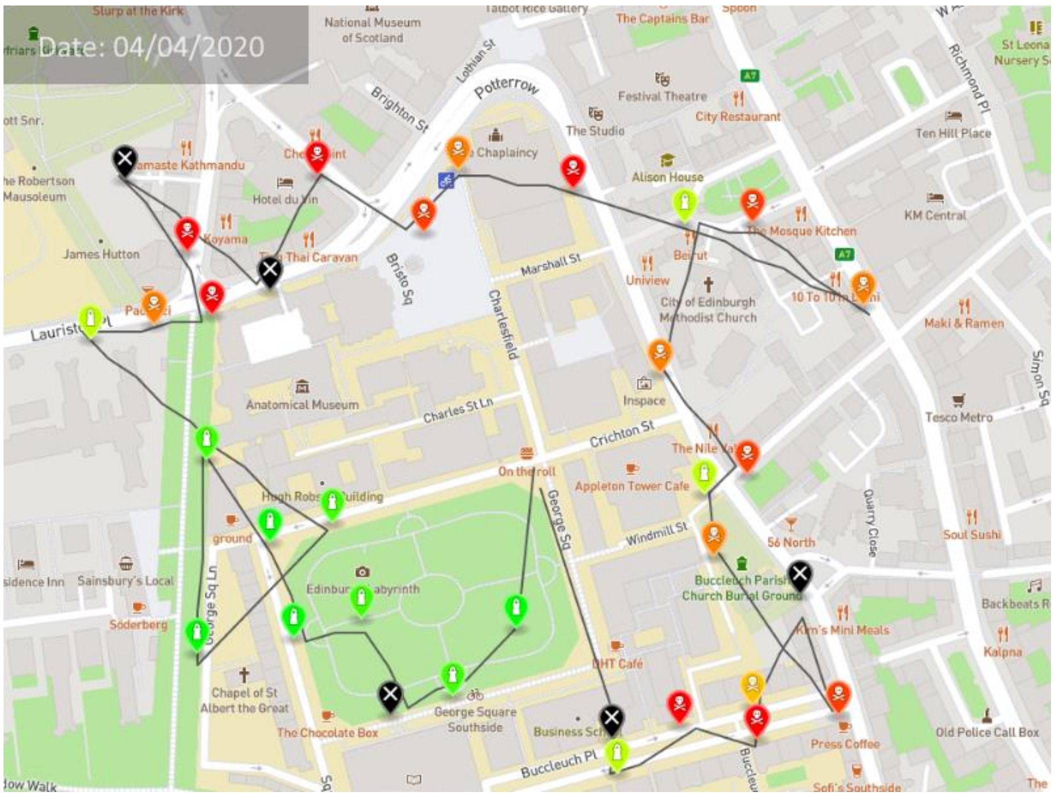
The most interesting part of the correction algorithm is its use of the previous move. The diagram shown below illustrates why this is necessary.

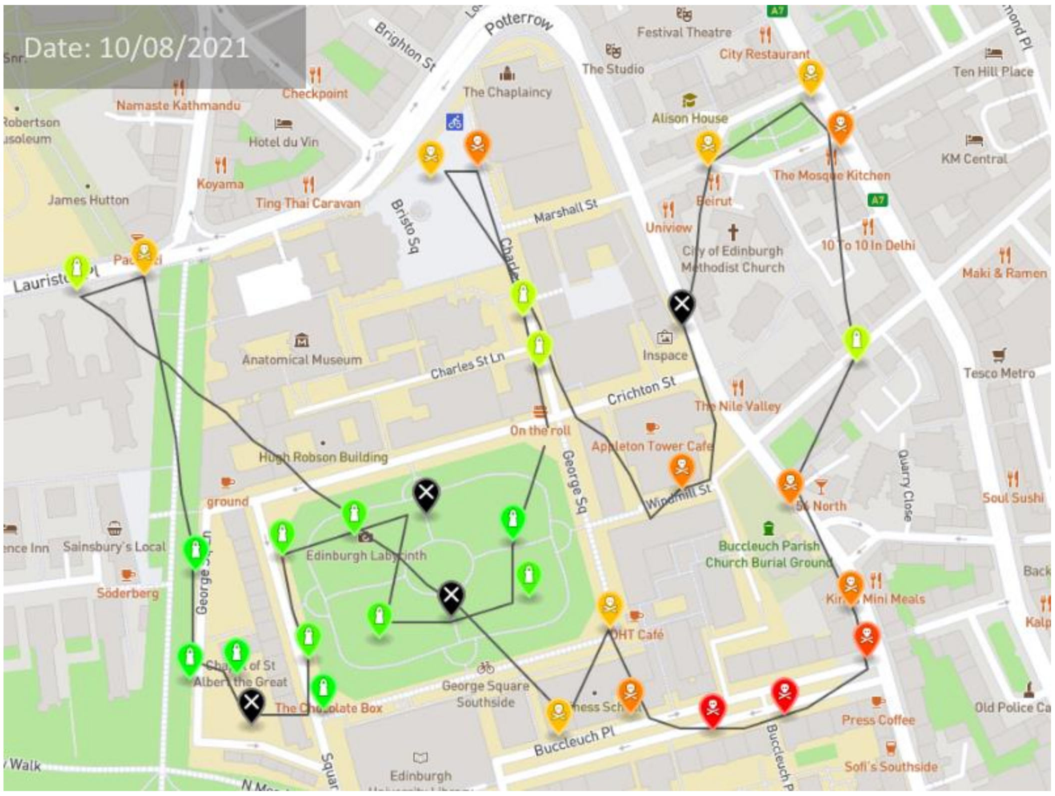

1. Drone makes optimal move
2. Drone calculates best angle to travel at
The 'Wall-scraping' Issue
● Current position
● Start/end position
3. Going back the way it came turns out to get it the closest to its goal
4. Drone calculates new best angle to travel at, which loops it back to the start

Storing our previous move fixes this issue by stopping the loop at step 2. Moving at the same angle (or closest to that angle) will free it of the wall and allow it to continue moving optimally. Sometimes we have to fly away from the target in order to reach it. With this ability to skirt around the no-fly zones, we ensure that we are generating valid paths of optimal (or close to optimal) distance.

The Results

Using the Sorted Edges algorithm in combination with FlightPaths allows us to return an ordered list of paths between each sensor on a given day. These are then outputted as flightpath text files that detail each move of the drone, and a GeoJSON map visually representing that complete path. You can see two such maps on the following page:





These demonstrate the success of the flight algorithm, connecting each point together in a logical order whilst avoiding no-fly zones such as the Informatics Forum or Appleton Tower. For these two paths in particular, the number of moves were 104 and 95 respectively, well below the 150 limit of the drone. The program executes in around 2 seconds, which is excellent. The visual output helps to confirm that the algorithm works as intended, satisfying the constraints and goals of the brief.