Software architecture

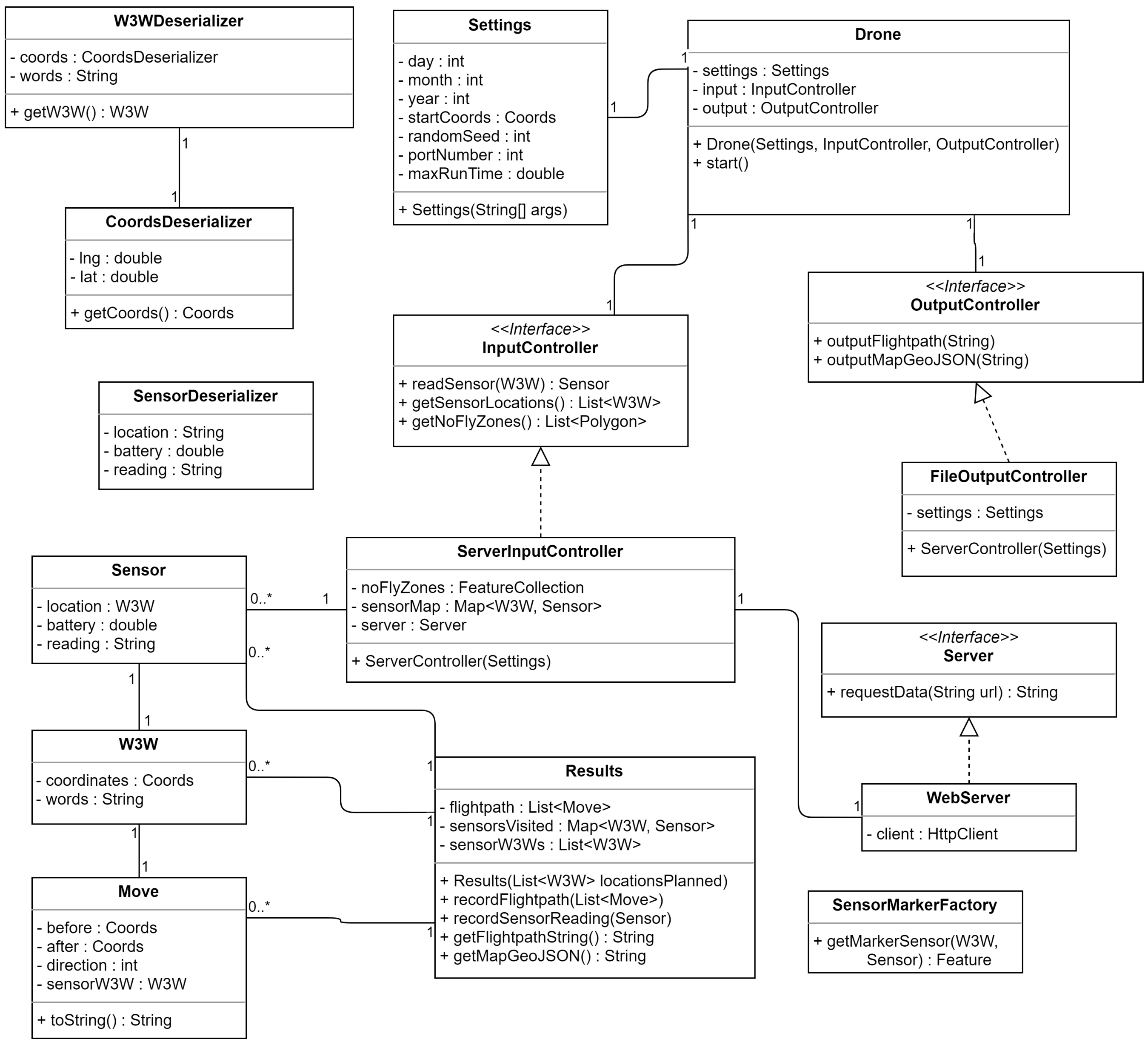

Figure 1: class diagram 1

This class diagram consists of the classes in the program which aren't directly related to the pathfinding algorithms (Sensor, W3W, and Move appear elsewhere, but I didn't want to clutter it to much), in packages aqmaps, and its subpackages io and deserializers. Note that when there are lines connecting the classes, I have also listed the connected class in fields for more clarity. This section also uses `Polygon` from the other half of the class diagram - note that this is my own class called `Polygon` and *not* `com.mapbox.geojson.Polygon`. In the class diagram I omitted getters whenever they were present, unless they were "special" - see `Desializers`.

IO package

The IO package contains 3 interfaces and their implementations that handle things relating to input and output of data. The interfaces have the goal of improving testability and allowing individual components to be easily swapped out for any future extensions of the application. One of the toughest things when doing the initial design was figuring out how to handle the fact that it was supposed to simulate a drone, while in a scenario very unlike what an actual would be doing in real life - for example the locations and readings of the sensors come in a single file from the server. I decided to deal with this interfaces, as explained below.

OutputController

I thought the way that the drone output its results would be something that might wanted to be changed, so I made an interface for it. At the moment the `FileOutputController` outputs files to the current directory, but it could also, for example send files back to a server, and to do this another class which implemented the interface could be written. I also thought that file output in general would be something good to be put in its own class, as it could be considered a separate unit of functionality. I also separated it from input because I though it made sense in general and so they could be swapped out independently. In addition, using an interface allowed me to pass in a fake `OutputController` to the drone which performs checks on the output for testing.

InputController

My reasoning for this is essentially the same as for the output. It makes sense as an individual unit, I want it to possible to swap it out in the future, and potentially helps with testing - though I ended up doing something else for that, see the next section. This class does not perform the actual communication with the server, which was delegated to `Server`, it performs all necessary processing relating to that - creating the requests and deserialising raw data into the objects that the rest of program will use. I also decided to keep stuff involving the mapbox library to the I/O classes (and `Results`), so that most of the program still functions if you wanted to use something else. This being an interface also allows you to make a certain drastic change - an implementation could connect `readSensor()` to the controls of an actual, physical drone, and "read the sensor" could also include "fly to the sensor" - this is one of the main ways I found to make the application as adaptable as possible to progress beyond the prototype stage.

Server

I wanted to keep the actual communication with the server, the HTTP client and stuff, in a separate class. I made an interface for it as I thought that it was something that could be swapped out - this also allowed you to swap just the `Server` and not the `InputController` if you still want its functionality. My original main idea for creating this interface was when I wanted to be test the program without needing to have the web server running. In my tests I pass in a fake server which gets data from the file system instead - this is also much quicker so it made mass testing much easier and faster.

Desializers package

This package consists of 3 classes which are used in the process of deserialising the JSON. The format of the classes in the JSONs was different to how I wanted them in the rest of the program, so I created these as temporary classes for the JSON to be deserialised into before processing them into the form that I actually wanted. The getters in two of these classes are not normal getters, they get the non-deserilised versions of the objects.

AQ maps package

The aqmaps package contains the other packages, but it also contains various classes that do not fit into any of the clearly defined other packages. They mostly consist of classes which function mostly as data types, and classes which run the highest level of operations that don't sense to be grouped together. I must also point out that the `App` class is missing from the diagram, but it doesn't really do much besides get the program started and isn't really part of what I would consider the class structure.

Drone

What this class represents is rather obvious. Despite its central importance it is actually a pretty short and simple class, but it brings things together, getting the input data, putting it into the flight planning algorithm, collecting data from the sensors, and sending the data off to be output.

Move, Sensor, Settings, and W3W

These are pretty self-explanatory classes which represent their namesakes. I can't think of anything in particular to say about them other than that they were pretty natural choices for classes given how clear it is what they represented.

Results

This is the class that holds the results of the algorithms - the flight path and the sensor data. I chose instead of doing this in `Drone` for example to have this as a class which holds the output data (flightpath and sensors), and processes it into the form which will be output. It was also a possibility to do this in `OutputController`, but I wanted that to focus on output specifically and also be able mock it for testing, so I made this separate.

SensorMarkerFactory

I wanted to have a class just called `SensorMarker` which subclassed the geojson `Feature`, however that was impossible since `Feature` does not have a public constructor so could not be extended. I instead chose to use this class which instead of representing the marker itself represents the thing that builds markers - this in fact makes a lot of sense because instances of the hypothetical `SensorMarker` wouldn't really be doing anything after being created, just being put into a FeatureCollection. This factory evidently handles the manufacture of sensor markers, such as setting the colour and symbol (using some code copied from heatmaps CW1), and I thought this class was the right one for my application since it was something with an easily separable and obvious purpose, and also because if you wanted to change how the markers were displayed, you would know exactly where to go.

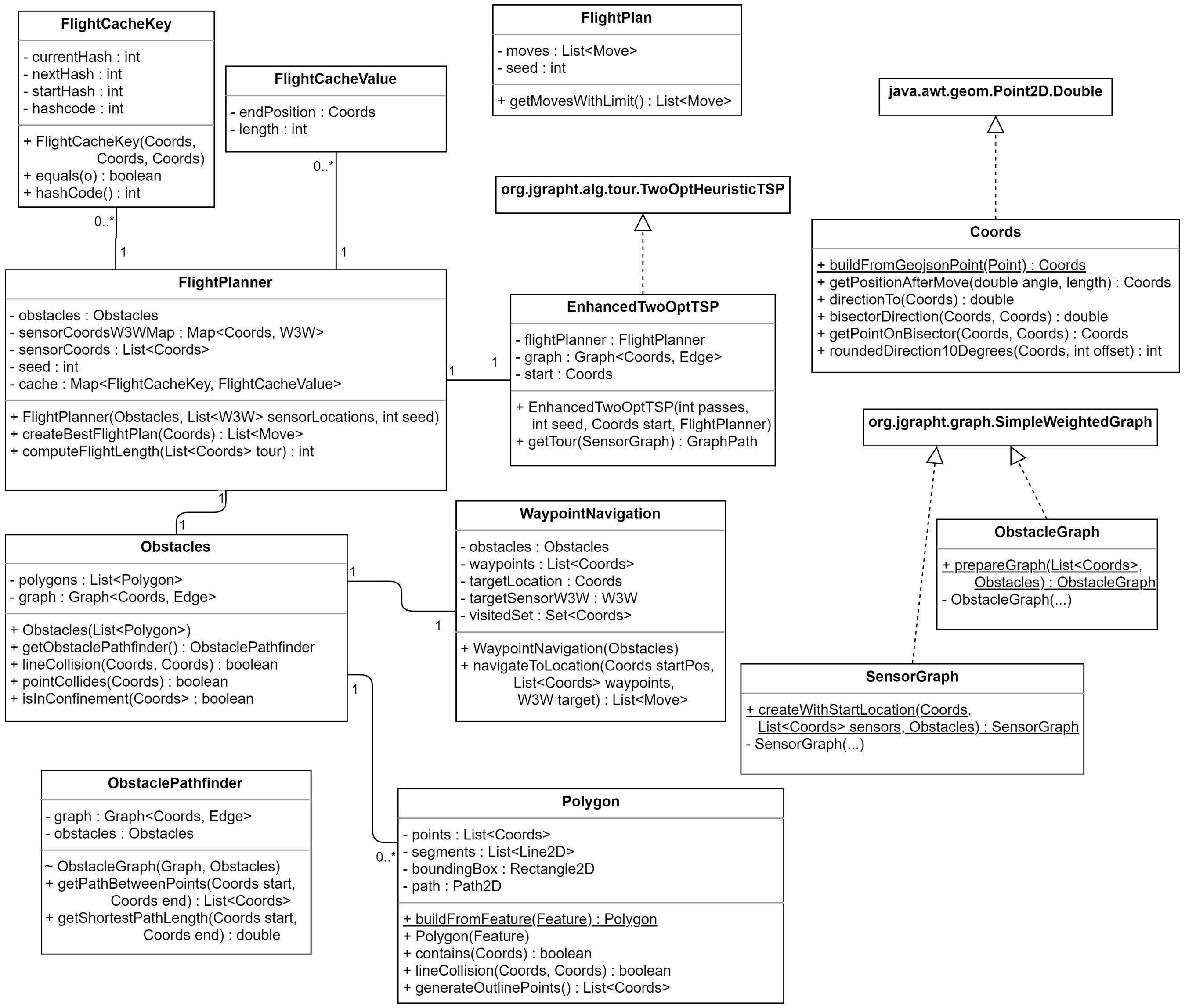
FlightCacheKey
- currentHash : int
- nextHash : int
- startHash : int
- hashcode : int
+ FlightCacheKey(Coords, Coords, Coords)
+ equals(o) : boolean
+ hashCode() : int
FlightCacheValue
- endPosition : Coords
- length : int
FlightPlan
- moves : List<Move>
- seed : int
+ getMovesWithLimit() : List<Move>
java.awt.geom.Point2D.Double
org.jgrapht.alg.tour.TwoOptHeuristicTSP
Coords
+ buildFromGeojsonPoint(Point) : Coords
+ getPositionAfterMove(double angle, length) : Coords
+ directionTo(Coords) : double
+ bisectorDirection(Coords, Coords) : double
+ getPointOnBisector(Coords, Coords) : Coords
+ roundedDirection10Degrees(Coords, int offset) : int
FlightPlanner
- obstacles : Obstacles
- sensorCoordsW3WMap : Map<Coords, W3W>
- sensorCoords : List<Coords>
- seed : int
- cache : Map<FlightCacheKey, FlightCacheValue>
+ FlightPlanner(Obstacles, List<W3W> sensorLocations, int seed)
+ createBestFlightPlan(Coords) : List<Move>
+ computeFlightLength(List<Coords> tour) : int
EnhancedTwoOptTSP
- flightPlanner : FlightPlanner
- graph : Graph<Coords, Edge>
- start : Coords
+ EnhancedTwoOptTSP(int passes, int seed, Coords start, FlightPlanner)
+ getTour(SensorGraph) : GraphPath
org.jgrapht.graph.SimpleWeightedGraph
ObstacleGraph
+ prepareGraph(List<Coords>, Obstacles) : ObstacleGraph
- ObstacleGraph(...)
Obstacles
- polygons : List<Polygon>
- graph : Graph<Coords, Edge>
+ Obstacles(List<Polygon>)
+ getObstaclePathfinder() : ObstaclePathfinder
+ lineCollision(Coords, Coords) : boolean
+ pointCollides(Coords) : boolean
+ isInConfinement(Coords> : boolean
WaypointNavigation
- obstacles : Obstacles
- waypoints : List<Coords>
- targetLocation : Coords
- targetSensorW3W : W3W
- visitedSet : Set<Coords>
+ WaypointNavigation(Obstacles)
+ navigateToLocation(Coords startPos, List<Coords> waypoints, W3W target) : List<Move>
SensorGraph
+ createWithStartLocation(Coords, List<Coords> sensors, Obstacles) : SensorGraph
- SensorGraph(...)
ObstaclePathfinder
- graph : Graph<Coords, Edge>
- obstacles : Obstacles
~ ObstacleGraph(Graph, Obstacles)
+ getPathBetweenPoints(Coords start, Coords end) : List<Coords>
+ getShortestPathLength(Coords start, Coords end) : double
Polygon
- points : List<Coords>
- segments : List<Line2D>
- boundingBox : Rectangle2D
- path : Path2D
+ buildFromFeature(Feature) : Polygon
+ Polygon(Feature)
+ contains(Coords) : boolean
+ lineCollision(Coords, Coords) : boolean
+ generateOutlinePoints() : List<Coords>
0..*
1
0..*
1
1
1
1
1
1
1
1
0..*

Figure 2: class diagram 2

This class diagram consists of the classes in the program related to the flight planning algorithm, in packages geometry, noflyzone, and flightplanning. Note that the `Coords` class is not connected to anything with lines, even it is used all over the place, because I wanted to avoid clutter.

Geometry package

Coords

Having a class which holds coordinate points or locations in space is pretty much a given for this application. `Point2D` was a natural class to consider using to do something like this, as opposed to using the point class in the mapbox library, or a making a class from scratch, because of inbuilt functions (like distance), and interoperability with other awt.geom classes like Line2D, Rectangle2D, and Path2D which have various desired methods. It may have been possible to use `Point2D` directly, and do calculations elsewhere but since I wanted to more calculations involving them than was available in `Point2D`, I decided this was where I should use subclassing and extend `Point2D.Double` to add my own methods. Note that it extends the double version specifically; `Point2D` is an abstract class - there is also the float version, but that doesn't use high enough precision for this project.

Polygon

The no-fly zones, which I often refer to as obstacles as is it easier and more general, come in the form of polygons. It is therefore natural that I would want to use a class to hold these polygons. Of course, mapbox already has a polygon class, but it provides no extra functionality beyond holding points, and isn't designed to be used for geometric calculations like this. I also as discussed earlier wanted to keep most of the program mapbox-free. Unfortunately, awt does not have a `Polygon2D` class so I was unable to use that or subclass it, so I made my own version based around the other awt 2D classes. An alternative to this class would be holding lists of various things directly in `Obstacles`, however this class is better for encapsulation, and makes it easier to do things polygon by polygon instead of keeping track of several lists - this is useful for collision checking which uses both bounding box and segments. Abstracting it away in this separate class also makes things easier to understand, and also maybe this class could be used in other places in the future.

No fly zone package

This package deals with the no-fly zones (obstacles).

Obstacles

This is the class which holds information about the obstacles, including the obstacle graph, and also provides obstacle pathfinders. I wanted to have a class which represented the obstacles in general, that could be passed around and used for both collision checking and pathfinding between points in various places in my program. I am using a quite complicated algorithm with multiple parts which all need access to these features, so it is convenient to have them easily accessible together in this class. This class and `ObstaclePathfinder` could potentially have been merged together, however that was not done for a reason, explained below.

ObstaclePathfinder

My program needs to have a way perform pathfinding, finding the shortest path which connects 2 points. Of course ideally this would just be a straight line, but often there will be an obstacle blocking the way, which we must find our way around. This is the class that encapsulates such functionality. There are 2 oddities with this class: why the functionality isn't just included in `Obstacles` instead, and why it is only accessible by the `getObstaclePathfinder()` method. As well as trying to be OOP-like this is necessary to allow the concurrent operation of the flight planning algorithm, as mentioned in algorithm 1. In the obstacle pathfinding algorithm (alg 4), it needs to make modifications to the obstacle graph, so to avoid illegal concurrent modification we must use clones of the graph. This is also why the constructor of this class has been explicitly made package-private

ObstacleGraph

My application uses JGraphT (https://jgrapht.org/) to assist with holding and running algorithms on graphs. This library was chosen as it simplifies development and allows use of optimised algorithms written by people who are much better programmers than me, and generally gives anyone working on this codebase less things to worry about. This class exists a subclass rather then using the `SimpleWeightedGraph<Coords><DefaultWeightedEdge>` directly because it is able to include more functionality (specifically the initialisation) inside the class, as well making it easier to differentiate between obstacle and sensor graphs. This class is not parameterized unlike its superclass, which comes with the added advantage of not needing to type `SimpleWeightedGraph<Coords><DefaultWeightedEdge>` all of the time.

Flight planning package

This package contains classes directly associated with how the drone flies around all of the sensors and returns to the start.

FlightPlanner

This class has the responsibility of taking the input locations of all off the sensors and the starting point, and producing an as short as possible flight plan. It is this package's point of communication with the rest of the program and responsible for high level coordination of the flight planning algorithm. This class was chosen since it has one clear purpose - plan the flight of the drone, and also because it needs a central location which handles the concurrency (see the algorithm section). Note that the flight plan length method is essentially the same (not in the real implementation, see the caption of alg) as the other method, as it essentially just computes a flight plan then gets its length. I think flight planner is good for a class, it sounds like job title with a clear role.

FlightCacheKey, FlightCacheValue

These wrap up the parameters and outputs for the flight planning cache, see the bottom of the description of algorithm 5.

FlightPlan

This wraps up a flight plan as a list of moves with the seed that generated it (see class documentation for why). Since this class already existed, it made sense to put another method in it (there are normal getters as well).

SensorGraph

This is the similar to `ObstacleGraph` but with sensors instead - it exists to be fed to 2-opt.

WaypointNavigation

This class is like a miniature version of `FlightPlanner`, not because it is smaller or more simple (in fact, could be considered the "core" of the algorithm), but because it exists to perform a specific subtask - navigate the drone from a starting position along a series of waypoints to a target. This exists as a separate thing to `FlightPlanner` because it is like its own self contained piece of functionality. It also carries a field `visitedSet` which is specific to this algorithm.

EnhancedTwoOptTSP

The 2-opt algorithm used in this application is not the usual one, it combines a normal call to JGraphT's 2-opt with a second pass using a modified version in this class to further improve it - hence the "enhanced". This class exists, and exists as a subclass of the class from JGraphT specifically, because it is a natural extension of the original, and so that it can call methods in its superclasses (including JGraphT's 2-opt as well as other utility functions in the abstract class above it). I think that this is the right class for my application since it implements a specific bit of functionality, while taking advantage of the fact that we are using OOP by extending a library class, which I quite like.

Class documentation

I have written a lot of comments and JavaDoc in the code which would be used by a developer if they were to actually maintain my code (the JavaDoc HTML is included in aqmaps.zip just in case, I know that it is not assessed). I tried using a tool to convert my JavaDocs into LaTeX, including private visibility items, and it came out as *seventy-five* pages - needless to say, what is included in this document is instead a much more concise summary, of mostly only public visibility.

Class App:

Flies a drone around Edinburgh to collect air quality data from sensors and create a map. This code follows the Google Java Style Guide at https://google.github.io/styleguide/javaguide.html **public static void main(String[] args)**ain method. args - a list of arguments in the form: day month year latitude longitude randomSeed portNumber [timeLimit]

Class Coords extends Point2D.Double:

Holds a longitude and latitude pair, using a Point2D. This class uses euclidean geometry and its calculations do not match with real life.

public static Coords buildFromGeojsonPoint(Point p)

Convert a mapbox point into a Coords

public double directionTo(Coords A)

Let this point be P. Calculates the direction or angle of the line PA with respect to the horizontal, where east is 0, north is pi/2, south is -pi/2, west is pi

public int roundedDirection10Degrees(Coords A, int offset)

Let this point be P. Calculates the direction of the line PA, rounded to the nearest 10 degrees, offset by an amount, and expressed in the range.

public double bisectorDirection(Coords A, Coords B)

Let this point be P. Calculate the direction of the acute bisector between the lines PA and PB.

public Coords getPositionAfterMoveDegrees(double degrees, double length)

Creates a new Coords which is the result of moving from the current location at the specified angle for the specified length. Angle in degrees version.

public Coords getPositionAfterMoveRadians(double radians, double length)

Creates a new Coords which is the result of moving from the current location at the specified angle for the specified length. Angle in radians version.

public Coords getPointOnBisector(Coords A, Coords B, double distance)

Let this point be P. Creates a point a specified distance away in the direction of the acute bisector between the lines PA and PB.

Class Drone:

Represents the drone. Performs route planning, than follows that plan to collect sensor data.

public Drone(Settings settings, InputController input, OutputController output)

public void start()

Start the drone and perform route planning and data collection for the given settings.

Class EnhancedTwoOptTSP:

A modified version of TwoOptHeuristicTSP from JGraphT which is used to compute tours which visit all sensors and returns to the starting point.

public EnhancedTwoOptTSP(int passes, int seed, Coords start, FlightPlanner flightPlanner)

public GraphPath<Coords,DefaultWeightedEdge> getTour(Graph<Coords,DefaultWeightedEdge> graph)

Computes a tour by first using JGraphT's TwoOptHeuristicTSP (the superclass of this) to find a short tour using the edge weights in the provided graph, which are straight line (obstacle avoiding) distance measures. Then, it runs a second pass of 2-opt to further improve upon the tour by instead using a FlightPlanner to generate the actual drone moves along the tour and using the number of moves as the weight of a tour.

public GraphPath<Coords,DefaultWeightedEdge> improveTour(GraphPath<Coords,DefaultWeightedEdge> graphPath)

(Code unchanged from library code other than type parameters) Try to improve a tour by running the 2-opt heuristic using the FlightPlanner to measure the length of tours.

Class FileOutputController implements OutputController:

Outputs to the current directory of the filesystem

FileOutputController(Settings settings)

(methods as inherited from interface)

Class FlightCacheKey:

A class which holds data about the input values of the flight planning algorithm from a position to a sensor, potentially with a next sensor. This is for use in the cache, so stores hashed value directly in order to save memory and time calculating extra hashes.

public FlightCacheKey(Coords startPosition, Coords currentTarget, Coords nextTarget)

public boolean equals(Object o)

Needed to work with a HashMap, automatically generated by IntelliJ.

public int hashCode()

Since this class stores the hashcode directly, we do not do any computation and just return it.

Class FlightCacheValue:

A class which holds data about the output values of the flight planning algorithm from a position to a sensor, potentially with a next sensor. This does not hold the actual tour, and is only used for the size of the tour, as it would use a lot of memory.

public FlightCacheValue(int length, Coords endPosition)

public int getLength()

0.0.1 public Coords getEndPosition()

Class FlightPlan:

A wrapper class for a flight plan as a list of moves, and the random seed that was used to generate it. This is needed so that results can be sorted by seed before finding the minimum, allowing for consistent operation even when concurrency is used.

public FlightPlan(int seed, List<Move> moves)

public List<Move> getMovesWithLimit()

Get the list of moves in the flight plan, limited to a maximum of 150 moves. In testing with the current possible input values, this never came close actually limiting the number of moves.

public int getSeed()

public List<Move> getMoves()

Class FlightPlanner:

Handles the creation of a flight plan for the drone. Uses JGraphT's TwoOptHeuristicTSP algorithm as part of process, which was the best performing of JGraphT's Hamiltonian Cycle algorithms, however this could be changed easily.

public FlightPlanner(Obstacles obstacles, List<W3W> sensorW3Ws, int randomSeed, double timeLimit)

Construct a flight planner with the given time limit in seconds. If the time limit is not greater than 0, turns it off and uses a maximum number of iterations instead.

public List<Move> createBestFlightPlan(Coords startPosition)

Create a flight plan for the drone which visits all sensors and returns to the start. Runs the algorithm a large number of times with different random seeds, in parallel, and chooses the shortest.

public int computeFlightLength(List<Coords> tour)

Computes the length of a flight plan which follows the given sensor coordinate tour.

Class public interface InputController:

Handles interaction with input from all remote information sources and devices. Gets information on sensor locations, no-fly zones, W3W locations, and sensor readings. Implementations of the interface can gather the information from any source, such as a simple web server for testing purposes, or from a full system where data is also read from real sensors.

Sensor readSensor(W3W location)

Reads information from the sensor at the provided W3W location

List<W3W> getSensorW3Ws()

Gets the list of sensors that need to be visited from a remote source

List<Polygon> getNoFlyZones()

Gets information about no-fly zones from a remote source

Class public class Move :

A class representing a single move to be made by the drone.

public Move(Coords before, Coords after, int direction, W3W sensorW3W)

getters for the same attributes as the in the constructor

Class ObstacleGraph extends extends SimpleWeightedGraph<Coords,DefaultWeightedEdge> :

A graph of the vertices of the obstacles and edges between them if they have line of sight.

public static ObstacleGraph prepareGraph(List<Coords> outlinePoints, Obstacles obstacles)

Prepare a weighted graph containing all points which form an outline around the polygons as vertices, and edges connecting them if they have line of sight, which have a weight equal to the distance between them. The graph uses outline polygons since if it used the original polygons, their points would occupy the same location and any line emerging from the corner of an obstacle would be considered to be colliding with it. See Polygon.generateOutlinePoints().

Class ObstaclePathfinder:

Handles obstacle evasion. Uses Obstacles and an ObstacleGraph to find paths between points which do not collides with any obstacles.

public List<Coords> getPathBetweenPoints(Coords start, Coords end)

Find the shortest path between the start and end points, navigating around obstacles if necessary.

public double getShortestPathLength(Coords start, Coords end)

Find the length of the shortest path between the start and end points, navigating around obstacles if necessary. Euclidean distance is used as the length measure.

Class Obstacles:

Holds information about the obstacles or no-fly zones that the drone must avoid.

public Obstacles(List<Polygon> polygons)

Constructs Obstacles out of a list of Polygons specifying their locations

public boolean isInConfinement(Coords point)

Determines whether or not a point is inside the confinement area

public boolean lineCollision(Coords start, Coords end)

Determines whether the line segment between the start and end points collides with a obstacle.

public boolean pointCollides(Coords coords)

Determine whether the given point is inside an obstacle, or outside the confinement area. This is currently only used in testing to generate random starting points.

public ObstaclePathfinder getObstaclePathfinder()

Gets an ObstaclePathfinder using these Obstacles. The ObstaclePathfinder uses a clone of the obstacle graph, allowing it to be used concurrently with other ObstaclePathfinder.

Class Interface OutputController:

Handles interaction with all output locations. Implementations may output to any source, such as to a file or to a server.

void outputFlightpath(String flightpathText)

Outputs the flightpath planned by the drone

void outputMapGeoJSON(String json)

Outputs the GeoJSON map containing the flightpath and the sensor readings collected by the drone

Class Polygon:

Holds a polygon as a list of the Coords that make up the vertices, in order

public static Polygon buildFromFeature(Feature feature)

Create a Polygon from a GeoJSON Polygon

public List<Coords> generateOutlinePoints()

Generates the points of a new polygon which contains the original by a very small margin. It generates points a distance of 1.0e-14 from each point in the original Polygon in the direction of the bisecting angle between the two adjacent sides, or the opposite direction if that point is inside the polygon. The resulting polygon will be larger than the original by a margin of 1.0e-14 on all sides.

public boolean lineCollision(Coords start, Coords end)

Determines whether the line segment between the start and end points collides with the polygon.

public boolean contains(Coords p)

Checks whether a given point is containing within this polygon.

getters

Class Results:

Holds and processes the calculated flightpath and collected sensor data

public Results(List<W3W> sensorW3Ws)

public void recordFlightpath(List<Move> flightpath)

Adds a calculated flight to the results

public void recordSensorReading(Sensor sensor)

Add a sensor with its readings to the results

public String getFlightpathString()

Gets a flightpath String of the following format: 1,[startLng],[startLat],[angle],[endLng],[endLat],[sensor w3w or null] 2,[startLng],[startLat],
w3w or null]

public String getMapGeoJSON()

public String getMapGeoJSON()

Class Sensor:

A sensor with a battery level and reading.

public Sensor(W3W w3wLocation, float battery, String reading)

public String getReading()

the reading of the sensor, as a String. If the battery level is 10

getters

Class SensorGraph extends SimpleWeightedGraph<Coords,DefaultWeightedEdge> :

A graph of the sensors and their distances to each other, taking into account obstacle evasion.

public static SensorGraph createWithStartLocation(Coords startPosition, Collection<Coords> sensorCoords, Obstacles obstacles)

Creates a complete weighted graph with the points of all of the sensors and the starting position. The edge weights are the shortest distance between the points, avoiding obstacles if necessary.

Class SensorMarkerFactory:

A factory which constructs sensor markers which displays the location, status and reading of a sensor. This factory does not produce hypothetical SensorMarker instances, but Feature instances instead, since Feature cannot be subclassed due to its lack of a public constructor.

public Feature getSensorMarker(W3W w3w, Sensor sensor)

Creates a marker located at the position of this sensor. If a sensor reading was taken successfully the marker is coloured and assigned a symbol based on the reading, and if it has low battery or was not visited, assigns different symbols.

Class Interface Server:

Handles requesting data from a server.

String requestData(String url)

Request the data that is located at the given URL. Will cause a fatal error if it cannot connect to the server, or if the requested file is not found.

Class ServerInputController implements InputController :

Implements the Remote interface using a connection to a simple web server.

public ServerInputController(Settings settings)

Create a new ServerInputController instance with the given settings, and collect and store data from server.

(methods as inherited from interface)

Class Settings:

Holds the settings derived from the command line arguments.

public Settings(String[] args)

(lots of getters)

Class W3W:

Holds what3words coordinate and word information.

public W3W(Coords coordinates, String words)

(getters)

Class WaypointNavigation:

A class which handles the navigation of the drone along a series of waypoints to a target. This is the core of the drone control algorithm that plans the movement of the drone itself including all rules about move lengths and directions.

public WaypointNavigation(Obstacles obstacles)

public List<Move> navigateToLocation(Coords startingPosition, List<Coords> waypoints, W3W targetSensorW3W)

Find a sequence of moves that navigates the drone from the current location along the waypoints to the target.

Class WebServer implements Server :

Handles requesting data from an HTTP server.

(methods as inherited from interface)

Drone control algorithm

The two main complicating factors in the design of the algorithm are the existence of the no-fly zones, and the constraints on the drone move length direction. Individually, these do not pose much of a problem, but it is together that they combine to create significant complexity, especially when not making assumptions. This algorithm attempts to capture the complexity of the situation, and any edge cases that may arise, and to aim for short flight paths by cutting corners literally, but not figuratively.

Algorithms

```
Input: the start coordinate position of the drone $c$, and $L$ a list of sensor locations
Output: a list of moves M that visits all sensors and return to the starting location
Function createPlan($c$, $L$)
    $G \leftarrow$ createSensorGraph($c$, $L$)
    T $\leftarrow$ enhancedTwoOptTSP($c$,$G$)
    rotate $T$ so that it starts and ends with $c$
    return constructFlightAlongTour($T$)

Input: a list of points $T$ starting and ending with the drone start position with the rest containing a tour of all of
       the sensors
Function constructFlightAlongTour($T$)
    $c \leftarrow T_0$
    $M \leftarrow [\,]$
    for $i \in [1, size(T) - 1]$ do
        $t \leftarrow T_i$
        if $i < size(T) - 1$ then
            $t \leftarrow$ cutCorner($c$, $T_i$, $T_{i+1}$)
        $W \leftarrow$ getPathBetweenPoints($c$, $t$)
        $M$.add(nagigateAlongWaypoints($c$, $W$, $1$, maxMoveLength($c$, $W_1$)))   // start at 1 since the first
         is the sensor we have already reached, see algorithm 6
        c $\leftarrow$ last(M)                          // update current location to the end of the latest move
    return $M$

Input: coordinate points c, t, n
Function cutCorner($c$, $t$, $n$)
    $\theta \leftarrow$ the direction which bisects the angle between lines $tn$ and $tc$                 // new direction
    $p \leftarrow$ point $0.634 * 0.0002$ degrees away from t in direction $\theta$                          // new target
    $min \leftarrow$ distance $ctn$
    if $cpn < min$ and not lineCollision($cpn$) then return $p$
    else return $t$
```

Algorithm 1: Creates a flight plan for the drone which visits all sensors and returns to the start. It first uses a two stage 2-opt heuristic on a pre-generated sensor graph to generate a tour. It then scans through the tour, constructing a flight plan along a list of waypoints which pathfind around any obstacles using the drone movement rules. Before each step of generating the flight plan, it attempts to "cut the corner", taking advantage of the fact that sensors have a 0.0002 range to shorten the route. The constant ratio 0.634 was chosen as it performed the best in testing: it is a trade-off between cutting more of the corner and being too close to the outside of the range and missing or overshooting (due to the constraints on drone movement). In the implementation there are many bonus features: the algorithm is run a large number of times, in parallel if possible, and the shortest flight plan is chosen. Depending on the options, this continues for a fixed number of iterations or a set amount of time (which can be input as an extra command line argument, default is no timer and run 40 iterations), after which it stops and chooses the best.

```
Input: the start coordinate position of the drone c, and L a list of sensor locations
Output: a graph G with vertices consisting of the start position and all sensors, with edge weight determined by
        the distance of the shortest path
Function createSensorGraph(c, L)
    G ← a new weighted graph
    G.addVertices(c + L)
    foreach pair of vertices (a, b) do
        G.addEdge((a,b), weight (euclidean distance) of getPathBetweenPoints(a, b))
```

Algorithm 2: This simple algorithm creates a graph for use by the TSP algorithms. Instead of using the euclidean distance between points in a straight line, it uses the obstacle pathfinder to calculate the length of the shortest route between them while also going around obstacles.

```
Input: a list of obstacles represented as polygons P
Output: a graph G with vertices consisting of the points surrounding all of the obstacles, and edges wherever they
        have line of sight with euclidean distance weight
Function prepareObstacleGraph(OP)
    L ← [ ]
    foreach p ∈ P do
        L ←the points of a polygon which forms a very narrow outline around P
    G ← a new weighted graph
    G.addVertices(L)
    foreach pair of vertices (a, b) do
        if not lineCollision(ab) then
            G.addEdge(a, b, distance(a, b))
```

Algorithm 3: Prepare a graph with all points which form an outline around the polygons as vertices, and edges if they have line of sight, which have a weight equal to the distance between them. The graph uses points from outlines of the obstacles since if it used the originals, points on the vertices would occupy the same location and the collision detection would consider any line emerging from the corner of an obstacle to be colliding with it.

```
Input: coordinate points a and b
Output: a path P of points which is the shortest path from a to b on graph O
Data: The obstacle graph O, as prepared by algorithm 3
Function getPathBetweenPoints(a, b)
    if not lineCollision(ab) then return direct path from a to b
    O.addVertex(a), O.addVertex(b)
    foreach vertex v ∈ O − {a, b} do
        if not lineCollision(av) then O.addEdge(a,v,distance(a,v))
        if not lineCollision(bv) then O.addEdge(b,v,distance(b,v))
    path ← dijkstraShortestPath(O, a, b)                          // from JGraphT
    O.removeVertex(a), O.removeVertex(b)               // (edges implicitly removed)
    return path
```

Algorithm 4: This algorithm finds the shortest path between the start and end points, navigating around obstacles if necessary. It does so by adding the start and end points to the obstacle graph, running Dijkstra's algorithm from the JGraphT library, and then removing the points so that it can be used again. I decided not to include pseudocode for Dijkstra's since it is library implementation of a common algorithm, that also uses some more complicated optimisations.

```
Input: the start coordinate $c$ and a graph $G$ with $n$ vertices with the start location and the sensors
Output: a tour $T$ which starts at $c$, visits all sensors and returns to $c$
Data: A list of polygons $P$, which represent the obstacles
Function enhancedTwoOptTSP($c$, $G$)
    T ← twoOptTSP($c$, $G$)
    repeat
        $l^{direct}_{original} \leftarrow$ direct straight line calculated length of $T$
        $l_{original} \leftarrow$ length(constructFlightAlongTour($T$))
        $change_{min} \leftarrow 0$
        $i_{min} \leftarrow -1$, $j_{min} \leftarrow -1$
        for $i = 0$, $i$++, while $i < n - 2$ do
            for $j = i + 2$, $j$++, while $j < n$ do
                $T_{temp} \leftarrow \text{applySwap}(T, i, j)$
                $l^{direct}_{new} \leftarrow$ direct straight line calculated length of $T_{temp}$
                if $l^{direct}_{new} < l^{direct}_{original} * 1.1$ then
                    $change \leftarrow$ length(constructFlightAlongTour($T$)) $- l_{original}$
                    if $change < change_{min}$ then
                        $change_{min} \leftarrow change$
                        $i_{min} \leftarrow i$, $j_{min} \leftarrow j$
        if $i_{min}$ and$j_{min} \neq -1$ then
            $T \leftarrow \text{applySwap}(T, i, j)$          // permantly make the change that improved the most
    until no move occurred
    return $T$
```

Algorithm 5: Computes a tour by first using JGraphT's TwoOptHeuristicTSP to find a short tour using the edge weights in the provided sensor graph, which are shortest path (obstacle avoiding) distance measures. Since that tour does not account for how the drone moves, it is likely that there we will be able to find a shorter tour if we took drone movement into account. However, running the drone flight planning algorithm every time we wanted to measure path weight in 2-opt would be too expensive since it cannot be stored as edge weights in a graph due to the nature of drone movement, especially when working through the initial stages with randomly generated tours which are very long and collide with a lot of obstacles.
Instead, we generate the tour with the simpler distance measure, and then run a second partial pass of 2-opt to further improve upon the tour by instead using constructFlightAlongTour() to generate the actual drone moves along the tour and using the number of moves as the weight of a tour. Since we know that the tour is quite good already, and that the drone flight path length and physical length are quite correlated, we further simplify the process by only considering swaps which do not increase the physical length of the tour by more than 10% - interrupting a roughly circular tour by criss-crossing to the opposite side while going around 3 buildings has no chance of making a shorter route.
In the implementation, twoOptTSP() is done with a single call to the library, and the rest of this function is implemented with a modified version of the JGraphT's source code implementation. Because of the various changes made to the conventional algorithm, I decided to present the pseudocode for the modified version here, but not the original. The constant value 1.1 was chosen since it performed the best in testing.
The real implementation does not use the same version of constructFlightAlongTour() (algorithm 1), it uses a modified version which is too similar to deserve its own pseudocode, but I still want to explain it here: First, instead of constructing the tour then finding the length, it just adds up the length as it goes along - that on its own makes little difference other than a potential tiny increase in speed. Since it is being called inside several layers of loops in the 2-opt algorithm, constructFlightAlongTour() is going to get called a lot, and it itself calls many other functions and is generally an expensive algorithm. The JGraphT version and other implementation of normal TSP do not have this problem as they can do a quick calculation using 4 values of a distance array instead of recalculating the length of the entire tour, but that is not possible here. Thinking about how 2-opt worked, I also noticed that calls to the function are often with rather similar tours, so it is potentially wasting time doing computations which it has already done. Going back to function constructFlightAlongTour(), I could see that the section inside the for loop could be seen as a function with input values c, T_i, T_{i+1}, and output values $c, length$, so I added a cache which maps those inputs to those outputs which is checked at the start of every loop iteration. I used a ConcurrentHashMap so it still worked with the parallelisation, and when running 40 iterations of createPlan() the ratio of misses to hits was about 1:3, and there was a speed up of 60-70%.

```
Input: the current coordinate position $c$, a list of waypoints $W$, the last of which is the target sensor or end position,
       the current waypoint number $i$, the number of moves remaining until we declare a potential path invalid $t$
Output: a list of moves which move the drone along the waypoints into range of the targets, in reverse order
$D \leftarrow ()$                                        // the set of points that have been examined so far
Function nagigateAlongWaypoints($c$, $W$, $i$, $t$)
    if $t = 0$ then
        return null
    foreach $offset \in [0, 10, -10, ..., 170, -170]$ do
        $\theta \leftarrow$ offset + direction from $c$ to $W_i$ rounded to the nearest 10°
        $d \leftarrow$ position after moving 0.0003 in the direction $\theta$
        if $d \in D$ then
            continue                                                       // try the next offset
        $D.add(d)$
        if the line cd collides with an obstacle then
            continue                                                       // try the next offset
        if $W_i$ is not the last waypoint and d has line of sight to $W_{i+1}$ then
            $M \leftarrow$ nagigateAlongWaypoints (d, W, $i+1$, maxMoveLength ($d$, $W_{i+1}$))
            if $M$ is not null then
                return $M.add($ a move from c to d in direction $\theta)$
            else
                continue                                                   // try the next offset
        if d is within 0.0002 of the target sensor or 0.0003 of the end position then
            return [Move(c, d, $\theta$, sensor if it is not the end)]
        $M \leftarrow$ nagigateAlongWaypoints (d, W, $i+1$, $t-1$)
        if $M$ is not null then
            return $M.add($ a move from c to d in direction $\theta)$
    return null                                                // if none of the offsets worked
Function maxMoveLength($c$, $w$)                                           // auxiliary function
    return $2 + \lceil cw \rceil / 0.0003$
```

Algorithm 6: Recursively finds moves which navigate from waypoint to waypoint, until the target location is reached. The main idea of this algorithm is to move in the direction of a waypoint until we reach it, and then move to the next waypoint, until we reach the target sensor or end position. If the target is an intermediate waypoint located on the corner of a no-fly zone, we determine that we have reached it not with but a range but when the next waypoint is in sight, since the waypoints exist to find paths around obstacles. We also keep track of which points we have examined so far to avoid looping, and stop examining a branch if it takes far more moves than expected, because it went the wrong way or got stuck somehow. If the move we tried hits an obstacle or gets stuck further down the recursive call, we try a different offset from the direction until we find one that works. In most cases, which do not need to avoid any obstacles, this algorithm will progress directly towards the target and terminate very quickly.

```
Input: coordinate points $a$ and $b$
Output: true if the line $ab$ collide with an obstacle, false otherwise
Data: A list of polygons $P$, which represent the obstacles
Function lineCollision($a$, $b$)
    if a or b not in confinement area then return true
    foreach $p \in P$ do
        if ab does not intersect the bounding box of p then
            if $\exists$ line segment $qr \in p$ such that qr and ab intersect then return true
    return false
```

Algorithm 7: A line which goes outside the confinement area is also considered to collide with an obstacle. This algorithm is called a very high number of times (1 million), so it is important that it is efficient. To strive towards this it first checks if it is inside the rectangular bounding boxes of the polygon (which has a highly optimised implementation in Point2D), and if it doesn't it can skip checking the line segments - and since most of the area is not close to an obstacle this works most of the time. Otherwise, it scans through all of the line segments of the polygon for line intersections using Point2D.intersectsLine(). At O(n) in the number of vertices of the obstacles polygons this is not the most asymptotically efficient algorithm for collision detection, and O(log(n)) algorithms do exist. However, considering that n is in this case very small, and the pruning that takes place, such algorithms are unlikely to offer significant speedup, so I chose to use this more simple one. However, they are one potential improvement that could be made, especially if much more complicated input obstacles were used.

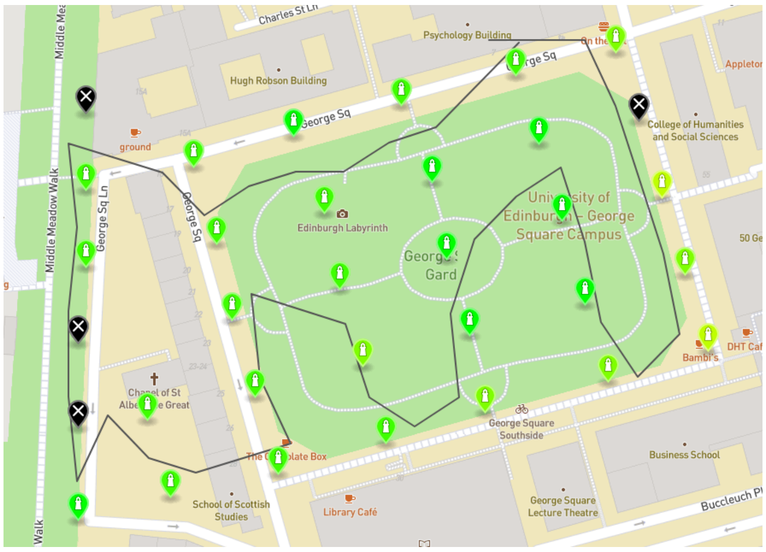
Figure 3: The output map from 01-04-2020. This is a very simple input, with all sensors close together and with no obstacles blocking anything. I chose this one since I thought that it demonstrated the corner cutting part of the flight planner well. At the very start, where the drone heads south-west, you can see that it isn't aiming directly at it, but instead cuts the corner to the next sensor while still entering the sensor range. It is then able to stay in the middle of the two rows of sensors, moving efficiently. Throughout the whole path you can see it sticking to the inside of the curve wherever possible.

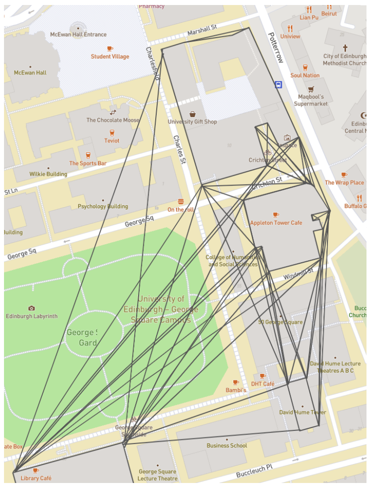
Figure 4: A depiction of the obstacle graph from algorithm 3

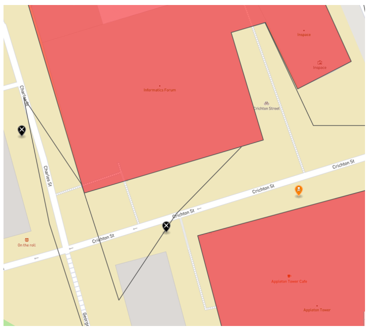
Figure 5: I noticed this particular part of one of the outputs and decided to add it as extra. This is an example of the algorithm dealing with a complex situation involving obstacles. The starting location is in the corner of the indent into the informatics forum, and the algorithm is able to navigate its way out of it. On the way back, it performs some very close calls with the building, passing about 7cm away from the corner of the informatics forum, before managing to almost crash into it again as comes to a stop right on the edge, but still within the end range.

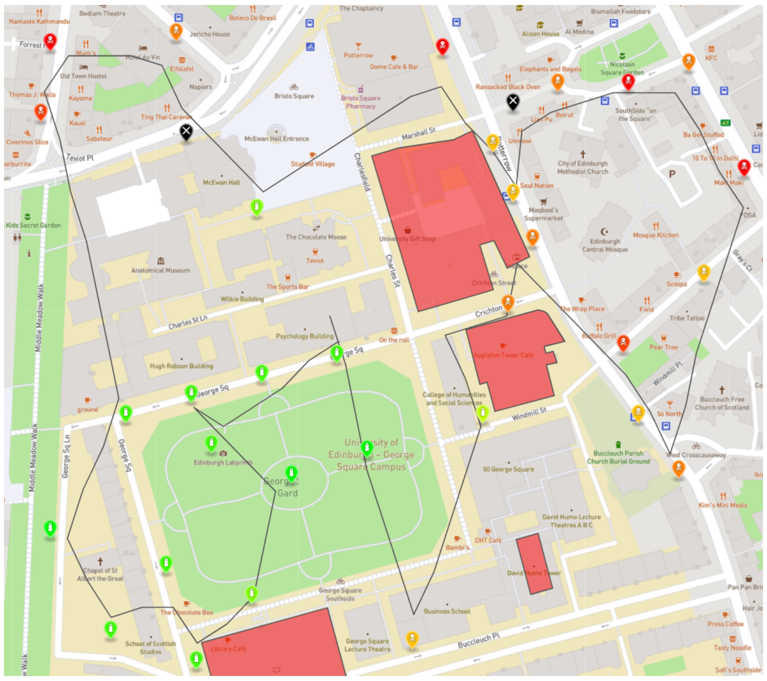
Figure 6: The results from an unknown day, because I lost it. This example again demonstrates the algorithm's corner cutting ability, as well as navigating around buildings, including through the gap. It also finds a nice shortcut in the gap area, where it almost hits the building, by reading the sensor through the corner and not getting stuck going in a circle to find its way back in.