Informatics Large Practical (ILP) Report

Nathan Sharp

December 04, 2020

1 Software Architecture description

This section provides a description of the software architecture of my application. My application is made up of a collection of java classes; this section will explain why I identified these classes as being the right ones for my application.

1.1 Entity Relationship Class Diagrams (ERDs)

ERDs for the project, including fields (f) and methods (m), are as follows,

Figure 1: Package **aqmaps** (root) Entity Relationship Diagram

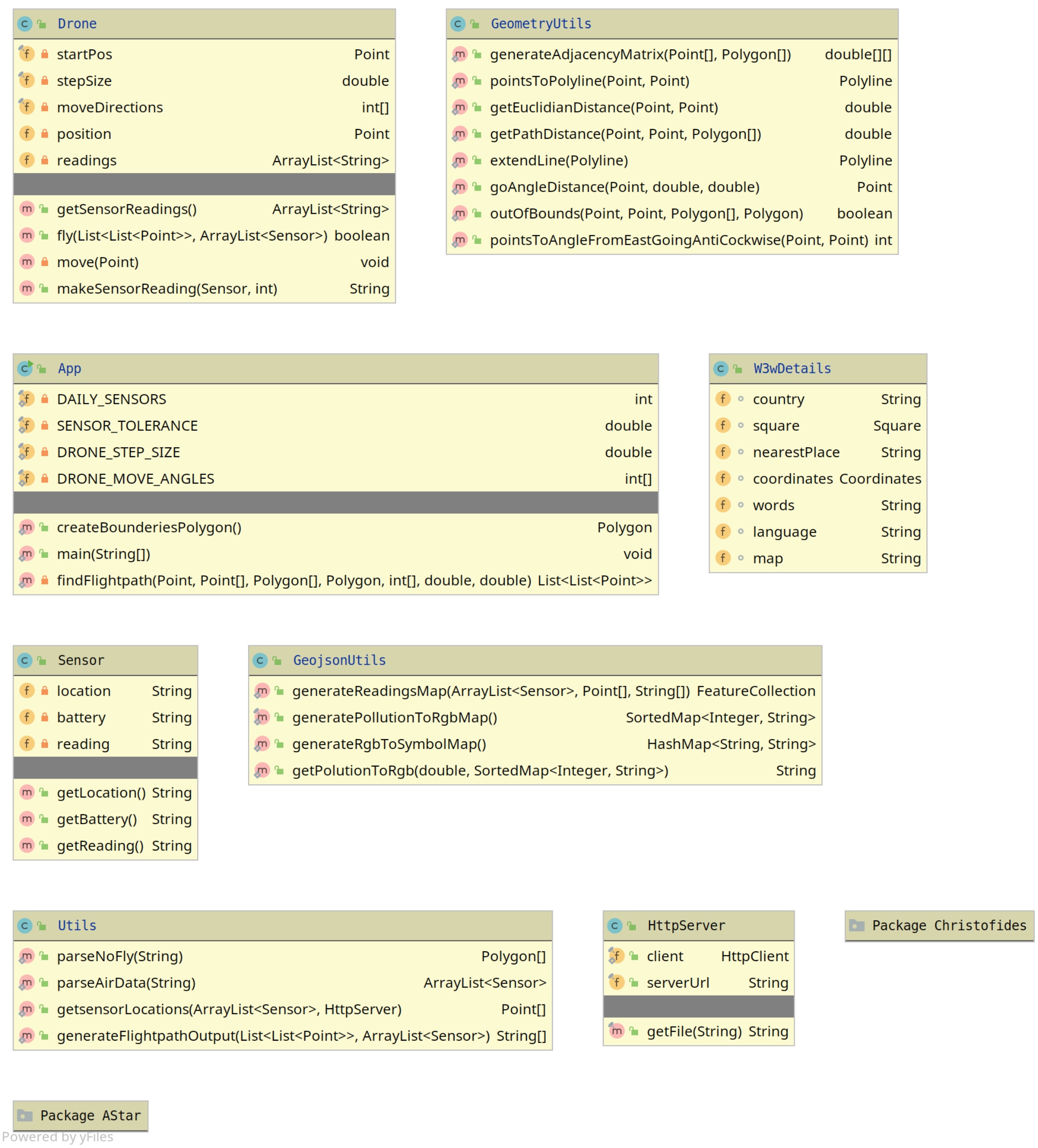


Figure 2: Package **AStar** Entity Relationship Diagram

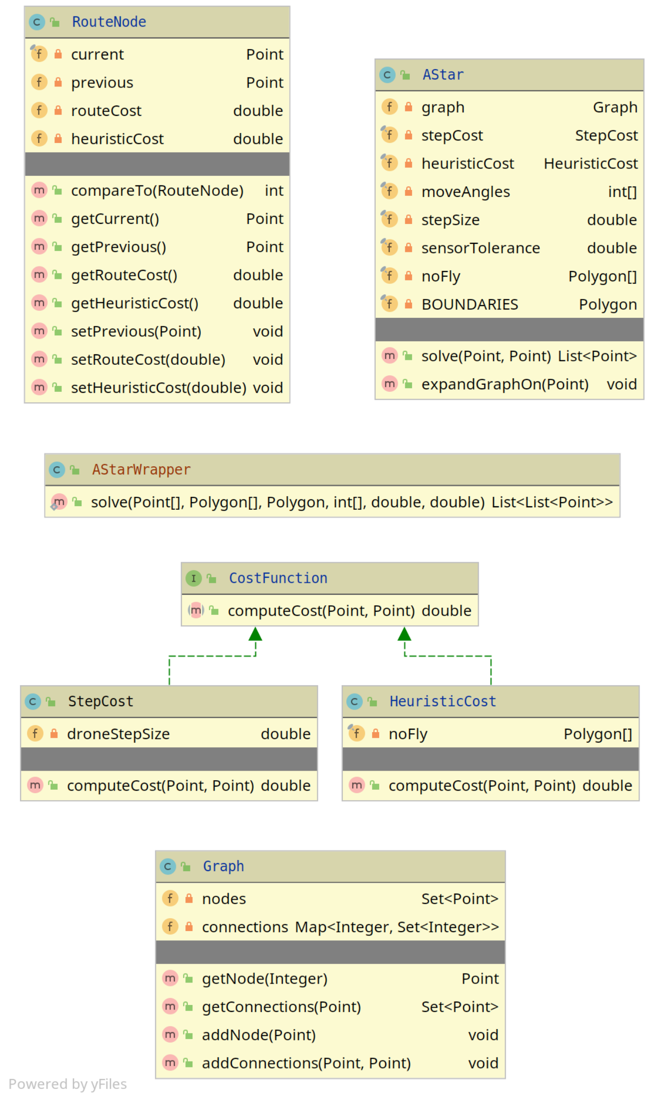
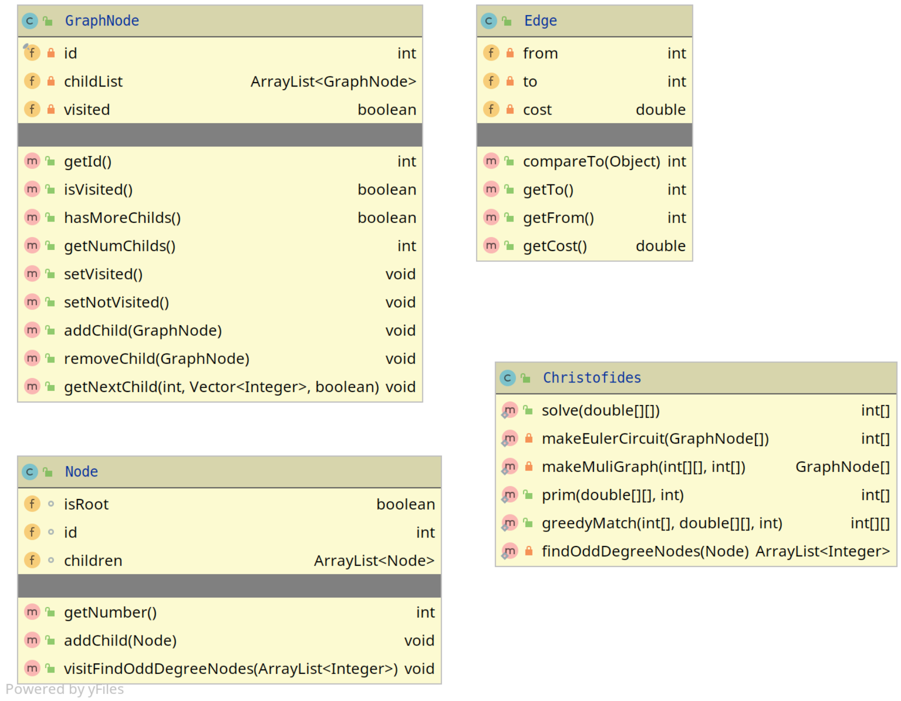


Figure 3: Package **Christofides** Entity Relationship Diagram



1.2 Explanation of classes

The source code, ER diagrams and javadoc documentation speak for my implementation in the best way I know how. It would, I think, be impractical and not particularly helpful to try too much to coerce the precision of its technical nature into prose. I offer only some brief highlights in into the though process experienced in creating this structure.

Within the *aqmaps* package files, *Sensor* and *W3wDetails* are Json deserialisation classes. *Drone* and *HttpServer* were obvious class choices, *GeometryUtils* was separated from *Utils* as I felt the former was a good collection of related utilities and *GeojsonUtils* was separated from *GeometryUtils* as they had name-space clashes between the two main packages used, com.esri.core.goemetry and com.mapbox.geojson. Hence this allowed the primary package in each case to be used without name qualification.

For the *AStar* package, at the highest level I chose to split *AStarWrapper* as a calling class for a more textbook implementation of *AStar* leaving the wrapper to deal with alot of the unique parts of the imple-menation. *CostFunction* is a simple interface for *StepCost* and *HeuristicCost* which allow raking of possible moves by distance traveled already and estimated remaining distance. *Graph* and *RouteNode* were the classes that were my choice for keeping track of the Algorithm's state.

Finally in the *Christofides* package files, most of the functionality is in the main file, and the three supporting classes provide essential data structure functionality.

In terms of code style the only language feature I actively avoided were streams even thought I do like their readability. I have a strong affinity for functional programming paradigms in general, however I do not personally enjoy their implementation in java as I find streams unnecessarily fiddly to set up and difficult to debug, although I am sure some of this is just my skill level lacking. Perhaps it would be nice if next year students were given the opportunity to program in Scala.

2 Class Documentation

Concise documentation for each class in my application.

The following javadoc is rendered to latex with makspll's javadoc-class-extractor (github).

2.0.1 uk.ac.ed.inf.aqmaps.Christofides

class Christofides

Class implemeting the Christofides Algorithm on an instance of the Traveling Salesman Problem

Extends **Object**

Members

public **Christofides**()	
public static int[] **solve** (**double**[][] AdjMatrix)	Entry point for the christofides algorithm
private static int[] **makeEulerCircuit** (**GraphNode**[] nodes)	Builds the union of mst and (bipartite) match, which is a multi graph
private static GraphNode[] **makeMuliGraph** (**int**[][] match , **int**[] mst)	Builds the union of mst and match, which is a multi graph
public static int[] **prim** (**double**[][] AdjMatrix , **int** dim)	Using Prim's algorithm to find the Minimal Spanning Tree.
public static int[][] **greedyMatch** (**int**[] prarentVec , **double**[][] AdjMatrix , **int** dim)	Finds a match between the nodes that hava odd number of edges. Not perfect but greedy, that is take the shortest distance found first. Then the next shortest of the remaining i chosen.
private static ArrayList<Integer> **findOddDegreeNodes** (**Node** root)	Finds vertexes which have odd number of edges.

class Edge

Class to model and edge on our graph

Implements **Comparable** Extends **Object**
implements Comparable

Members

public **Edge** (**int** from , **int** to , **double** cost)	Constructs an Edge for a graph
public int **compareTo** (**Object** edgeObj)	
public int **getTo**()	
public int **getFrom**()	
public double **getCost**()	

class GraphNode

GraphNode class for christofides algorithm. Keeps track of its id, children and prviously visited nodes

Extends **Object**

Members

public **GraphNode** (**int** id)	Constructor for Graphnode
public int **getId**()	
public boolean **isVisited**()	
public boolean **hasMoreChilds**()	
public int **getNumChilds**()	
public void **setVisited**()	
public void **setNotVisited**()	
public void **addChild** (**GraphNode** node)	
public void **removeChild** (**GraphNode** node)	
public void **getNextChild** (**int** goal , **Vector<Integer>** path , **boolean** firstTime)	

class Node	
Node class for christofides algirithm	
Extends **Object**	
Members	
public **Node** (**int** id)	Constructor for node class, setting chidren to null and root to false by default
public **Node** (**int** id , **boolean** isRoot)	
public int **getNumber**()	
public void **addChild** (**Node** node)	
public void **visitFindOddDegreeNodes** (**ArrayList<Integer>** oddNodes)	

2.0.2 uk.ac.ed.inf.aqmaps.AStar

class AStar	
Class to solve a pathfinding instance between two Points using A* search	
Extends **Object**	
Members	
public **AStar** (**Graph** graph , **StepCost** stepCost , **HeuristicCost** heuristicCost , **Polygon[]** noFly , **Polygon** BOUNDARIES , **int[]** moveAngles , **double** stepSize , **double** sensorTolerance)	
public List<Point> **solve** (**Point** from , **Point** to)	Finds an A* omptimal route between two points given the constraints of the system
public void **expandGraphOn** (**Point** node)	Expands graph frontier arround ‘node‘ for each angle in ‘moveAngles‘ at distance ‘stepSize‘

class AStarWrapper	
Class implementing the the totality of our A* search algorithm to plot a roundtrip on a set of ‘vertics‘ by calling the AStar between every progressive pair of points	
Extends **Object**	
Members	
public **AStarWrapper**()	
public static List<List<Point>> **solve** (**Point[]** vertices , **Polygon[]** noFly , **Polygon** BOUNDARIES , **int[]** moveAngles , **double** moveSize , **double** vertexTolerance)	Entry point to solve the A* instance

class Graph	
Graph class for A* search algorithm containing a set of ‘nodes‘ and map of ‘connections‘	
Extends **Object**	
Members	
public **Graph** (**Set<Point>** nodes , **Map<Integer, Set<Integer>>** connections)	Constructor creating a graph from nodes and connections
public Point **getNode** (**Integer** id)	
public Set<Point> **getConnections** (**Point** node)	
public void **addNode** (**Point** node)	Adds a node to the graph
public void **addConnections** (**Point** from , **Point** to)	Adds connection to graph

class HeuristicCost

Class to compute the Heuristic cost to the goal point

Implements **CostFunction** Extends **CostFunction**

Members

public **HeuristicCost** (**Polygon**[] noFlPolygons)	
public double **computeCost** (**Point** from , **Point** to)	Description copied from interface: CostFunction

class RouteNode

Class to model geographical nodes on A* route
kepping track of the ‘previous‘ node, ‘routeCost‘, the cost so far & ‘heursticCost‘,
the estimated to goal node

Implements **RouteNode** Extends **RouteNode**

Members

RouteNode **RouteNode** (**Point** current)	Constructor for a route node setting the previousNode to null, the routeCost to infinity and the heuristicCost to infinity
RouteNode **RouteNode** (**Point** current , **Point** previous , **double** routeCost , **double** heuristicCost)	
public int **compareTo** (**RouteNode** other)	
public Point **getCurrent**()	
public Point **getPrevious**()	
public double **getRouteCost**()	
public double **getHeuristicCost**()	
public void **setPrevious** (**Point** newPrevious)	
public void **setRouteCost** (**double** newrouteCost)	
public void **setHeuristicCost** (**double** newheuristicCost)	

class StepCost

Class to model a complex step function, in our case it is constant

Implements **CostFunction** Extends **CostFunction**

Members

public **StepCost** (**double** droneStepSize)	
public double **computeCost** (**Point** from , **Point** to)	Description copied from interface: CostFunction

2.0.3 uk.ac.ed.inf.aqmaps

class App

Informatics Large practial (ILP): Coursework 2.

Program to create a flightpath and fly imaginary drone round the Edinburgh University campua collecting sensor readings and avoiding obstacles.

Algorithms Utilised: 1) Christofides (TSP): for aproximation of an effient sensor circuit tour 2) A* Search: for generating routes between sensors avoiding buildings 2.1) A* Heuristic: distance (incliding round obstalcle) to target.

Extends **Object**

Members

public **App**()	
public static Polygon **createBounderiesPolygon**()	Creates the Map sqaure boundary box for our the drone over the specified area of the edinbugh university campus
public static void **main** (**String**[] args)	Entry Point to the the "aqmaps" drone flying program
private static List<List<Point>> **findFlightpath** (**Point** startPoint , **Point**[] sensorLocations , **Polygon**[] noFly , **Polygon** BOUNDARIES , **int**[] DRONE_MOVE_ANGLES , **double** DRONE_STEP_SIZE , **double** SENSOR_TOLERANCE)	Calcutes a move by move flightpath circuit for the drone to follow round the sensors (params defined in constructor)

class Drone

Class to represent the functionality of our drone which flys round the point on our route

Extends **Object**

Members

public **Drone** (**Point** startPos , **double** stepSize , **int**[] moveDirections)	Constructor for drone class
public ArrayList<String> **getSensorReadings**()	
public boolean **fly** (**List<List<Point>>** flightpath , **ArrayList<Sensor>** sensors)	Fly the drone round its daily sensor flightpath
private void **move** (**Point** to)	Makes a single move
public String **makeSensorReading** (**Sensor** sensor , **int** index)	Reads a sensor, when in range

class GeojsonUtils

Class for geojson related utility fuctions

Note com.mapbox.geojson classes can used without qualification (they clash with com.esri.core.geometry which usually take precedence)

Extends **Object**

Members

public **GeojsonUtils**()	
public static FeatureCollection **generateReadingsMap** (**ArrayList<Sensor>** sensors , **Point**[] sensorLocations , **String**[] flightpath)	Creates geojson map of sensors & flightpath (as per Coursework spec.)
public static final SortedMap<Integer, String> **generatePollutionToRgbMap**()	Initialises a pollution-value to rgb-hexcode map for colorcoding sensors in the output geojson where (key, value) = (pollution-bucket-upper-bound, RGB-color-string) as specified in ilp-coursework.pdf Figure 5
public static HashMap<String, String> **generateRgbToSymbolMap**()	Initialises rgb-hexcode to picture-symbol map for attaching symbols to points in output geojson (as specified in ilp-coursework.pdf Figure 5)
public static String **getPolutionToRgb** (**double** concentration , **SortedMap<Integer, String>** pollutionToRgb)	Maps pollution concentration estimate to RGB colour bucket

class GeometryUtils

Geometry utility functions

Extends **Object**

Members

public **GeometryUtils**()	
public static double[][] **generateAdjacencyMatrix** (**Point**[] vertices , **Polygon**[] noFly)	Generates an Adjacency Matrix for an array of points accounting for the shortest path round a noFly object
public static Polyline **pointsToPolyline** (**Point** point1 , **Point** point2)	Constructs a polyline from points
public static double **getEuclidianDistance** (**Point** point1 , **Point** point2)	Calculates the euclidian distance between two points
public static double **getPathDistance** (**Point** point1 , **Point** point2 , **Polygon**[] noFly)	Gets the path distance between two point in the map taking the shortest path round a noFly object if necessary.
public static Polyline **extendLine** (**Polyline** line)	Extends input line 0.01 units in both directions
public static Point **goAngleDistance** (**Point** point , **double** angle , **double** distance)	Calculates new point, an angle and distance from an existing point (angle goes clockwise from positive x axis)
public static boolean **outOfBounds** (**Point** point1 , **Point** point2 , **Polygon**[] noFly , **Polygon** BOUNDARIES)	Checks if the line between two points is out of bounds (main-BOUNDARIES/no-fly-zones)
public static int **pointsToAngleFromEastGoingAntiClockwise** (**Point** point1 , **Point** point2)	Calculates the nearest integer angle between 2 point from East (negative x-axis) going anticlockwise

class HttpServer

HttpServer class for connecting to and acessing the files form a local server

Extends **Object**

Members

public **HttpServer** (**String** serverUrl)	
public final String **getFile** (**String** filepath)	Function Description

class Sensor

Sensor class to hold deserialised Json representation

Extends **Object**

Members

public **Sensor**()	
public String **getLocation**()	
public String **getBattery**()	
public String **getReading**()	

class Utils

Utility functions

Extends **Object**

Members

public **Utils**()	
public static Polygon[] **parseNoFly** (**String** noFlyGeoJson)	Parse no flybuildings from geojson to com.esri.core.geometry Polygons
public static ArrayList<Sensor> **parseAirData** (**String** airDataJson)	Parse air data to ArrayList
public static Point[] **getsensorLocations** (**ArrayList<Sensor>** sensors , **HttpServer** server)	Retrieve sensor location via what3words (w3w) server file
public static String[] **generateFlightpathOutput** (**List<List<Point>>** flightpath , **ArrayList<Sensor>** sensors)	Generates flightpath output as per Coursework specifications

class W3wDetails
What3words (w3w) details class to hold deserialised Json representation
Extends **Object**
Members
public **W3wDetails**()

3 Drone Control Algorithm

This section explains the algorithm which is used by my drone to control their flight around the air-quality sensors and back to the start location of their flight, while avoiding all of the n-fly zones

3.1 Top Level

At the top level the algorithms used to control the drone and their call structure are shown in the code snippet below (App.java>findFlightpath(), lines 162-183).

```
1
2      // generate Adjacency matrix of distances between points
3      // accounting for shortest path round no-fly object
4        double[][] adjacencyMatrix =
5            GeometryUtils.generateAdjacencyMatrix(pointsToVisit, noFly);
6
7        // find an (semi) optimal round trip pointsToVisit ordering
8        // using the Christofides algorithm
9        int[] routeIndices = Christofides.solve(adjacencyMatrix);
10
11       // reorder points optimally using Christofides routeIndices
12       Point[] orderedPoints = new Point[pointsToVisit.length+1];
13       for(int i=0;i<orderedPoints.length-1;i++) {
14         orderedPoints[i] = pointsToVisit[routeIndices[i]];
15       }
16       // make starting point also as the finishing point
17       orderedPoints[orderedPoints.length-1] = startPoint;
18
19       // generate a flightpath for our round-trip using A* search
20       List<List<Point>> flightpath = AStarWrapper.solve(orderedPoints,
21        noFly, BOUNDARIES, DRONE_MOVE_ANGLES, DRONE_STEP_SIZE, SENSOR_TOLERANCE);
22
23       return flightpath;
```

Listing 1: Top level algorithm call structure

Details on the implementation and theory of–

- `generateAdjacencyMatrix`'s distance function (line 5)
- `Christofides`'s algorithm (line 9)
- `AStarWrapper`'s A* search implementation (line 20)

are covered in the following three sections.

3.2 `generateAdjacencyMatrix`'s Distance Function

Our distance function takes into account the no fly zones by looping over each no-fly building and altering the standard euclidean distance in the following way if the vector in question intersects with the no-fly building (GeometryUtils>getPathDistance() lines 79-100).

```java

        /* distance += outer path round smaller side of a polygon */

        // extend line
        Polyline extendedLine = extendLine(line);

        // split polygon into left cut and right cut
        GeometryCursor cuts = OperatorCut.local().execute(false,
         noFlyConvHull, extendedLine, null, null);
        Polygon leftCut  = (Polygon) cuts.next();
        if ( leftCut == null ) { continue; }
        Polygon rightCut = (Polygon) cuts.next();

        // calculate the preimeter of each cut
        double leftCutPerimeter  = leftCut.calculateLength2D();
        double rightCutPerimeter = rightCut.calculateLength2D();

        // get intersection lengeth
        double intersectionLength = OperatorIntersection.local()
         .execute(noFlyConvHull, line, null, null).calculateLength2D();

        // update path distance
        pathDist += Math.min(leftCutPerimeter,rightCutPerimeter)
                        - intersectionLength;
```

Listing 2: Calculating the true distance between two points

Note that the no-fly buildings are approximated as its convex-hull so as to guarantee a simple 2 point intersection with the line.
Overall I think it was a strong algorithmic decision and a clean implementation to calculate the true distances before considering an optimal route.

3.3 `Christofides` Algorithm

To find an optimal order to visit the sensors in I used the Christofides algorithm. The Christofides algorithm is a famous approximation Algorithm for the Traveling Salesman Problem (TSP). The Traveling Salesman Problem is that of finding a minimum distance tour of 'cities', (aka. nodes on a graph with weights between them) that visits each city exactly once. Its a famously hard NP problem meaning an exact solution cannot be found in polynomial time. This means even for our input of 33 sensors it is a bad idea to brute-force the solution especially if your hardware is limited– as might be the case if running locally on a drone.

The following code outlines the sub-process' for my implementation of the Christofides algorithm (Christofides.Chrisofides> (lines 22-35).

```java
    // calculate an mst using prims algorithm
    // mst = minimum spanning tree
    int mst[] = prim(AdjMatrix, AdjMatrix[0].length);

    // find a matching between between the nodes of odd-degree
    // (even number by the handshake lemma)
    int match[][] = greedyMatch(mst, AdjMatrix, AdjMatrix[0].length);

    // build the union of mst and match (a multigraph)
    GraphNode nodes[] = makeMuliGraph(match, mst);

    // calculate final route as an euler tour
    // possible as every vertex now has an even degree
    int route[] = makeEulerCircuit(nodes);   // removes loops
```

Listing 3: Christofides TSP approximation algorithm implementation

Note (obviously) full implementation of all sub-process' in the source code.

This is a a very strong choice of algorithm for finding an optimal sensor route as it finds a tour at most $1.5\times$ longer than the optimal route. In fact, the Christofides algorithm was the best known polynomial-time approximation of the TSP until this year (2020) when a new algorithm with an approximation ratio of $1.5 - 10^{36}$ was discovered.

Note that github users dsrahul30 and faisal22 have implementations I used as reference points for my design.

3.4 `AStar` Search Implementation

A* search is a simple but powerful graph finding algorithm that find efficient path between points on a graph. The important idea separating A* from a more simple graph search functions (such as naive BFS) is the idea of a 'hueristic-cost' which is used in combination with the total cost of reaching a node to expands the path in the least expensive direction. We chose to use the true path between a node (candidate drone position) and the target, as seen in listing 2 as our heuristic function.

We used A* to calculate the path to each individucal vertex (sensor) o our graph. Combined with a quick check to test if our done would be stepping over a no-fly zone (and removing the edge if so), this was all we needed to implement our drone navigation.

This this is a simple but strong choice of algorithm. 1 drawback is that it might not be highly adaptive to flying a drone in the real world and there is only so much complexity that can be added to A*.

Credit to geeksforgeeks.com, baeldung.com and brilliant.org for inspiration on the specifics of my implmentation.

4 Flightpath Renders

This section contains 2 graphical figues which have been made using the http://geojson.io website, rending the flight of my drone.

Figure 4: Drone Flightpath Render 1: 15/06/2021

Figure 5: Drone Flightpath Render 2: 01/01/2020