

# Informatics Large Practical Coursework 2

s1803764

## Table of Contents

1. Software architecture description.....	p1
2. Class documentation.....	p3
3. Drone control algorithm.....	p8

# SOFTWARE ARCHITECTURE DESCRIPTION

## Project description

To program a drone to fly around and collect sensor readings from sensors in a defined confinement area. This drone can only move in steps of 0.0003 degrees and can only turn in degrees of 10. This drone must also avoid no-fly-zones in the area whilst mapping its route. After collecting readings from all the sensors this drone must return to its start point and output the readings of the sensors (to a .geojson file) and a log of the movements it made during its flight (to a .txt file).

## Software architecture summary

### 1) Retrieving command line inputs

This program works by first retrieving all the relevant command line arguments that specify the date we want to map a route for, the starting point (in longitude and latitude), and the port of connection. Upon retrieval these inputs are validated for data type and correctness (eg. does not allow impossible dates). If one of these inputs is invalid, the program terminates.

### 2) Connecting to the web server

Once these inputs are retrieved and validated our program first tries to connect to the web server at the specified port using the java HTTP class. If this fails, the program terminates.

### 3) Retrieving the relevant data from the web server and parsing it into Java objects

#### 3.1) Retrieving and parsing the maps file

Assuming a successful connection to the web server our program retrieves the data for all the sensors to be read for the specified date. It does this by retrieving the *air-quality-data.json* file for the given date (in the *maps/YYYY/MM/DD/* directory) which stores the data for all the sensors: the What3Words location, the battery percentage, and the air-quality reading. These are parsed using substring indexing and are stored in custom 'Sensor' objects.

#### 3.2) Retrieving and parsing the What3Words files

Now that we have the sensor data, we must retrieve the respective coordinates for each What3Words location. Our program does this by iterating through all the different sensors to retrieve the *details.json* file for each given What3Words location (in the *words/W1/W2/W3/* directory). The only data we need from this file is the 'coordinates' object which represents the centre of the respective What3Words tile. These are parsed using substring indexing and are stored in their respective 'Sensor' objects. All of the sensors are then stored in one global ArrayList of Sensor objects called 'sensors'.

#### 3.3) Retrieving and parsing the no-fly-zones file

The last thing we need to retrieve from the web server is the no fly zones for the drone, these represent tall buildings which the drone must avoid in order to prevent a crash. Our program does this by retrieving the *no-fly-zones.geojson* file (in the *buildings/* directory). This file contains a Geo-JSON FeatureCollection which stores each no-fly-zone as a feature. The only data we need from each of these features is the 'Polygon' geometry object which stores an array of coordinates representing the vertices for the given no-fly-zone. These are parsed using substring indexing and are stored in custom NoFlyZone objects. All the no-fly zones are then stored in one global ArrayList of NoFlyZone objects called 'noFlyZones'.

#### 4) Find optimal sensor route

We use an algorithm to determine the optimal route to visit sensors. I implemented many different algorithms to find the best based on both performance and execution time.

#### 5) Calculate and record valid moves for the drone to follow this route

We start at the coordinates specified by the input arguments. Our program then iterates through the *findPoint()* method which returns a valid move that takes the drone closer to the next sensor in the queue. This is iterated until the drone has visited all the sensors in the route. These moves are calculated using planar trigonometry and will be discussed in detail further on.

Java classes used:

- `Java.io.FileWriter`
- `Java.io.IOException`
- `Java.util.ArrayList`
- `Java.util.Arrays`
- `Java.net.URI`
- `Java.net.http.*`

## CLASS DOCUMENTATION

### Object classes

Note that **all** the variables for each of the following objects are private and non-static. Private because they can be accessed via getter methods, and non-static as no default values are used meaning an object instance is required. Along with this, **all** class methods are non-static as they all require data from object instances. Class methods are only made public if they are accessed outside of their native class.

***\*NOTE:** getter and setter methods will be omitted from each class' 'Methods' description. If a given class has no 'Methods' description this is because it has no methods.*

### Fragment

This is a custom class for the *temperate()* algorithm. Simply put, this object represents an edge between 2 sensors. Find a full explanation of its implementation by reading the description of the *temperate()* algorithm in the 'Drone control algorithm' section.

#### Variables:

Sensor sensor: represents the original sensor to be expanded upon.

Double avgDist: represents the mean distance 'sensor' is from all the other sensors for the given day.

Sensor bestDestSensor: represents the best sensor for the drone to visit after 'sensor' (closest).

### Point

This object represents geographical coordinates.

#### Variables:

Double lng: represents the longitude coordinate

Double lat: represents the latitude coordinate

#### Methods:

Boolean isEqual(Point pointA)

Checks whether the given point instance is equivalent to pointA. Returns true if so, false otherwise.

Boolean checkConfinement()

Checks whether the given point instance is within the confinement area. Returns true if so, false otherwise.

### NoFlyZone

This object represents a no-fly zone.

#### Variables:

ArrayList<Point> points: represents the vertices of the given no fly zone.

## Sensor

This object represents an air-quality sensor.

### Variables:

`String location`: represents the What3Words location of the given sensor.

`Double battery`: represents the battery reading of the sensor.

`Double reading`: represents the air-quality reading made by the sensor.

`Point point`: represents the geographical coordinates of the sensor.

### Methods:

`String getReadingColour()`

Air-quality classification method which returns an rgb-string based on the given sensor's air-quality reading.

`String getReadingSymbol()`

Air-quality classification method which returns the name of a symbol based on the given sensor's air-quality reading.

## LineGraph

This object represents a straight-line function. It is extremely useful and effective for calculating whether a given drone path goes across a no-fly zone boundary. A full description of this utilisation can be found in the description for 'Avoiding no-fly zones and staying within the confinement area' in the 'Drone control algorithm' section.

### Variables:

`Double gradient`: represents the gradient of the function.

`Double yint`: represents the y-intercept of the function.

`Point p1`: represents the first point that was used to define the given straight-line function.

`Point p2`: represents the second point that was used to define the given straight-line function.

## Move

This object represents the movement of a drone between two points. This is very useful when calculating moves in *findNextMove()* as it allows us to easily store/log all of our drone's movements.

### Variables:

`Point origin`: represents the point of the drone at the beginning of the move.

`Point dest`: represents the point of the drone at the end of the move.

`Double angle`: represents the angle of movement from 'origin' to 'dest'.

## Route

This object represents a given drone route. I decided to represent a route as an object as it allows us to neatly store all the relevant data and functions. This not only promotes good readability but also gives us the ability to easily map many different routes and compare their results.

### Variables:

`ArrayList<Sensor> sensorRoute`: represents the optimized route of sensors for the drone to follow.

`ArrayList<Point> pointRoute`: represents the entire route the drone takes.

`ArrayList<Sensor> unreadSensors`: represents the sensors the drone has not visited yet.  
`String dataGeojson`: represents geo-json code which stores the markers for all of the sensors as Point features and the drone route as a LineString feature.

`String flightpathTxt`: represents the contents of the drone movement log file. Each line represents a move and stores data in the following order; the move number, coordinates before the move, angle of movement, coordinates after the move, and the What3Words location of the sensor visited in that move ("null" if none).

#### **Methods:**

`void findMoves()`

Given the starting point and route of sensors to follow, this method iterates through the `findNextMove()` function which takes the drone's current position and finds the optimal next position given what sensor needs to be visited next. Each of these found moves are then appended to the log file variable `flightpathTxt`, and every time a sensor is visited a classification marker is appended to the geojson variable `dataGeojson`. This is iterated until either all the sensors have been read or the drone has completed 150 moves. If the drone did not manage to visit all the sensors then the sensors that were not visited are appended to `dataGeojson` as blank markers.

`void writeOutputFiles()`

This method outputs the route's resulting geojson and text files in the current working directory.

## **Method classes**

Note that **all** the variables and methods for the following method classes are static. This is due to the fact they are not treated as objects and thus evidently do not rely on data stored in object instances. The only data they need are retrieved via public variable/method calls and input arguments. Again, class methods are only made public if they are accessed outside of their native class.

## **Algorithms**

This class is used to store all the sensor route optimisation algorithms.

#### **Methods:**

`ArrayList<Sensor> greedy()`

`ArrayList<Sensor> temperate()`

`ArrayList<Sensor> twoOpt(ArrayList<Sensor> sensorRoute)`

`ArrayList<Sensor> swap(ArrayList<Sensor> sensorRoute)`

`Sensor getClosestSensor(Sensor sens, ArrayList<Sensor> sensorRoute)`

`ArrayList<Sensor> getClosestSensors(Sensor sens)`

## **FileReading**

This class is used to store all the methods used for parsing web server files.

#### **Methods:**

`ArrayList<Sensor> parseJsonSensors(String fileContents)`

`Point parseJsonW3Wtile(String fileContents)`

`ArrayList<Sensor> getSensorCoords(ArrayList<Sensor> inputSensors)`

`ArrayList<NoFlyZone> parseNoFlyZones(String fileContents)`

## FileWriting

This class is used to store all the methods that are used to write to files.

### Variables:

String endFeatureCollectionGeojson:

String startMarkerGeojson:

String startLineStringGeojson:

### Methods:

String getGeojsonMarker(Sensor sens, Boolean beenVisited)

String getGeojsonRoute(ArrayList<Point> route)

void writeToFile(String filePath, String fileContents)

## GeometricalCalcs

This class is used to store all the geometrical calculation methods.

### Methods:

Double calcDistance(Point p1, Point p2)

Double calcAngle(Point origin, Point dest)

Point transformPoint(Point origin, Double angle)

## MoveCalcs

This class is used to store all the single move calculation methods.

### Variables:

Move lastMove:

Point lastSensorPoint:

### Methods:

Move findNextMove(Point currPoint, Point nextPoint)

Boolean isMoveRedundant(Move current, Point currSensorPoint)

Boolean isPointInRange(Point destination, Point actual)

Double calcRouteCost(ArrayList<Sensor> sens)

Double calcEdgeCost(Point origin, Point dest)

Double calcActualDist(Point origin, Point destination)

Boolean isPathValid(Point origin, Point dest)

Boolean checkNoFlyZones(Point p1, Point p2)

Boolean isIntersection(LineGraph path, LineGraph bound)

## Webserver

This class stores the methods which allow us to connect to the web server and retrieve files.

### Variables:

String wsURL:

HttpClient client:

**Methods:**

```
void initWebserver()  
String getWebserverFile(String path)
```

**Main Class (App.java)**

Note that like the previous method classes all variables and methods here are static. This main class is where we keep all the input arguments, constants, and web server data, thus most variables here are made public as all classes need access to this data. The only private variables being 'randomSeed' (as it is unused) and 'sensorRoute' (as if it is needed by a function it is passed as an argument). All methods besides the main are private as they are not utilized in any other classes.

**Variables:**

```
final double maxLat  
final double minLat  
final double maxLng  
final double minLng  
double errorMargin  
final double pathlength  
ArrayList<NoFlyZone> noFlyZones  
ArrayList<Sensor> sensors  
String dateDD  
String dateMM  
String dateYY  
Point startPoint  
int randomSeed  
String portNumber  
ArrayList<Sensor> sensorRoute
```

**Methods:**

```
void checkDateIsValid(String day, String month, String year)  
Integer checkIsNumber(String date, String name)  
void getSensorData()  
void getNoflyzoneData()  
void findOptimalRoute()
```



# DRONE CONTROL ALGORITHM

## Finding the optimal sensor route

In order to decide the optimal sensor route I decided to try lots of different algorithms to find what was optimal in this context.

### Algorithms used

After experience with the travelling salesman problem I had a good idea of what algorithms I wanted to try use for this drone.

Initial route setting algorithms: these set a route based purely on distances between sensors.

- **Greedy**

The Greedy algorithm works by iterating through the sensors the drone needs to visit and chooses a route based on which sensor is closest to the last.

The route is initialized with the first sensor in the list. Thereafter route expansion is done by adding the next closest available sensor. This process is continued until a complete route is formed.

- **Temperate (custom algorithm)**

I wanted to create an algorithm that finds optimal paths by prioritizing the sensors which have the highest mean distances from other sensors. This prioritization works by creating fragments/edges (represents the transition between two sensors) between the sensor with the highest mean distance and expanding it with its best possible path (the closest available sensor). This is iterated for all sensors from highest to lowest mean distance and placed in a priority queue (descending order of mean values). To prevent redundancy in this priority queue we only allow a given sensor to be in a maximum of two fragments (as a single sensor can be connected to a maximum of two other sensors). Once a priority queue of fragments/edges is found then the route can begin to be created. The route is initialized with the first fragment in the priority queue. Thereafter route expansion is based upon availability in the priority queue. By this I mean that based upon the given sensor we need to expand (last sensor in the current route), the algorithm first checks if this sensor can be found in a fragment in the priority queue. If so, the other sensor from this given fragment is added to the route. Otherwise, the algorithm finds the best available transition for the given sensor and adds it to the route. This process is continued until a complete route is formed. In order to ensure maximum efficiency redundant fragments (fragments which contain sensors that are not available) are deleted when found upon each iteration.

This algorithm almost works in the opposite way to the greedy algorithm in which rather than prioritizing using the shortest distances possible, mine prioritizes **not** using the longest distances possible, this is why I decided to name this algorithm 'Temperate' (the antonym of greedy).

Route refinement algorithms: these naively switch points in the route and see if this improves the overall route cost (total distance travelled).

- **Swap heuristic**

The Swap heuristic algorithm works by swapping adjacent sensors in the route to see if it improves the cost.

This algorithm iterates through each element in the given route to try all possible adjacent swaps. If a single swap in the loop is successful (improves the overall cost) we iterate through the route again (because the route has been changed). If no successful swaps are made throughout an entire loop the algorithm is terminated.

- **2-Opt heuristic**

The 2-Opt heuristic algorithm works by flipping the path between two sensors in the route to see if it improves the cost.

This algorithm uses a nested loop in order to get two indexes that represent sensors in the route. For every iteration we then try reverse the path between these two sensors. If a single reversal from the entire nested loop was successful (improves the overall cost) we iterate through the route again (because the route has changed). If no successful path reversals are made throughout an entire nested loop the algorithm is terminated.

**\*\*NOTE:** All of these algorithm implementations can be found in *Algorithms.java* and can be easily swapped between by using the *findOptimalRoute()* function in *App.java*

## Calculating distances between sensors

Calculating the distances between sensors accurately is a very critical part of producing an effective algorithm in this context. This is evident as all our route optimisation algorithms use distance to map a route with the smallest overall distance and thus minimal number of moves.

Typically we would just be able to measure the direct distance between two points, however, given our drone has to avoid no-fly zones this makes a direct distance not very reliable as there could be a no-fly zone between these points.

To solve this problem, when calculating the distance between sensors I would check if the path between these sensors intersected any of the no-fly zone boundaries. If so, I would then map out the actual distance the drone would have to follow to go around these no-fly zones. This was done by iterating *findMove()* until we were in range of the destination sensor, the resulting distance would just be the number of moves multiplied by the path length. Otherwise, if there was no obstruction between these sensors we would just take the direct Euclidean distance.

This was an extremely effective factor in optimizing the route as it maximizes both the performance and accuracy of our algorithms.

## Calculating moves

Given we want to minimize the total number of moves our drone takes, calculating moves efficiently is just as or even more important than finding an optimal sensor route. This is evident as our drone must be able to navigate its way between sensors in an optimal way in order to maximize efficiency

whilst conforming to its constraints on movement length and direction.

## Calculating a single move

This is all done in the *findMove()* method.

To calculate a single move, we must first know the current point and destination point (destination sensor coordinates) of the drone. With these points we will then be able to calculate the angle between our drone and the destination sensor using planar trigonometry:

$$\text{angle} = \tan^{-1}(m_{AB}) , \quad \text{where } m_{AB} = \text{gradient between points A \& B}$$

Although this angle is most likely not in degrees of 10 it allows us to find the next best options. Given this movement direction constraint we realize our drone must zigzag towards the sensor. To do this we can just take the nearest angles on either side of the real angle:

$$\text{newAngle} = (\text{realAngle} - (\text{realAngle} \% 10)) \pm 10$$

We can then use these two angle options to transform the current point and thus find the two possible next points. If an invalid point is found (outside confinement area or crossing a no-fly zone) the angle is changed until the point is valid. We then choose our move based on which point is closest to the destination point (given that this move is not redundant; discussed below).

A problem with this approach is when our drone needs to go around a no-fly zone. This is because it is likely our drone will have to move further away from the destination sensor in order to get around the given no-fly zone. This would evidently not work given the way our algorithm chooses the next move (based on which point is closest to the destination sensor) and thus would end up getting caught in an infinite loop of redundant moves. To prevent this, after every call of *findMove()* we store the move and destination sensor in global variables. Then when choosing which point is best we first check each of the moves are not redundant using the *isMoveRedundant()* function, and then compare distances. The *isMoveRedundant()* function works by checking if the angle of the input move is opposite of the last move (  $|\text{angle}_{\text{input}} - \text{angle}_{\text{lastMove}}| = 180$  ) and if the input destination sensor is the same as the last destination sensor. If these are both true, then we know this input move is redundant, and thereby we would choose the other available move in *findMove()*.

## Avoiding no-fly zones and staying in the confinement area

In order to avoid no-fly zones and stay in the confinement area when calculating a move in *findMove()* I created the *isPathValid()* method. This method takes the current point and the next point as inputs.

Using the *checkConfinement()* method it first checks if the 'next point' is within the confinement area by ensuring its coordinates are within a range of latitudes and longitudes defined by the global constants *maxLat*, *minLat*, *maxLng*, and *minLng*.

Next, using the *checkNoFlyZones()* method it checks if the path from the current point to the next point intersects any of the buildings/no-fly zones. It does this by iterating through each boundary for all of the no-fly zones and sees if the drone path intersects any of these boundaries using the *isIntersection()* function. If an intersection is found the loop is broken and the path is returned as invalid (false).

This *isIntersection()* function works by expressing this drone path and the given boundary of the no-fly zone as straight-line functions. Then using simple coordinate geometry, we can see if these lines intersect at a point in the range of the given boundary.

*isIntersection()*: calculating an intersection between the LineGraph *path* and LineGraph *bound* inputs (returns true if there is no intersection)

- First we declare the necessary variables:  
 $m_{net} = m_{path} - m_{bound}$  , where *m* represents the gradient  
 $c_{net} = c_{bound} - c_{path}$  , where *c* represents the y intercept  
 $maxLat = \text{MAX}(bound.point1.lat, bound.point2.lat)$   
 $minLat = \text{MIN}(bound.point1.lat, bound.point2.lat)$   
 $maxLng = \text{MAX}(bound.point1.lng, bound.point2.lng)$   
 $minLng = \text{MIN}(bound.point1.lng, bound.point2.lng)$
- Then we see if there was any intersection  

```
//Case if the path is a vertical line
if mpath = ±∞ and path.point1.lng ≤ maxLng and path.point1.lng ≥ minLng then return false

//Else checks that there is an intersection by seeing that the net gradient is not zero (not parallel lines)
else if mnet ≠ 0 then
    lngintersection =  $\frac{c_{net}}{m_{net}}$ 
    latintersection = mpath * lngintersection + cpath

    //Checks if the coordinate of this intersection is on the boundary of the no-fly zone
    if (lngintersection ≤ maxLng and lngintersection ≥ minLng) or
       (latintersection ≤ maxLat and latintersection ≥ minLat) then return false

//If no value has been returned yet there must be no intersection
return true
```

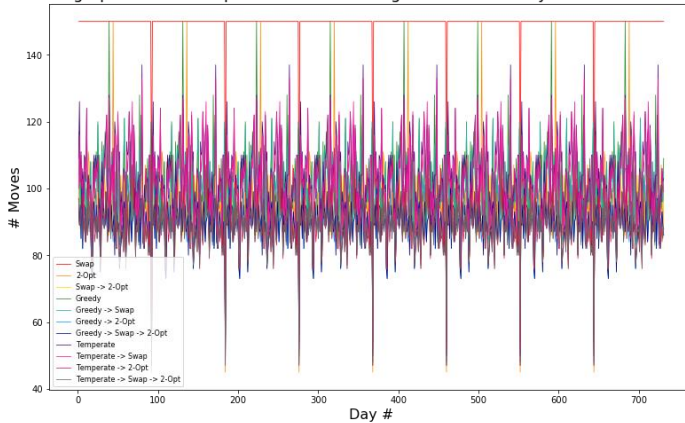
## Testing

Algorithms	Performance Metrics (over 731 maps)					
	Avg. # moves	Best # moves	Worst # moves	Standard deviation	Avg. execution time (s)	Time complexity
Swap	149.42	89	150	NA	0.108071135	O(n)
2-Opt	95.48	45	150	10.54	0.236662107	O(n <sup>2</sup> )
Swap -> 2-Opt	94.73	48	113	8.55	0.29001368	O(n <sup>2</sup> + n)
Greedy	101.93	63	150	11.56	<b>0.017783858</b>	O(n <sup>2</sup> )
Greedy -> Swap	99.42	53	121	10.43	0.02872777	O(n <sup>2</sup> + n)
Greedy -> 2-Opt	88.83	48	<b>102</b>	<b>7.03</b>	0.149110807	O(n <sup>2</sup> )
Greedy -> Swap -> 2-Opt	88.90	50	104	7.09	0.151846785	O(n <sup>2</sup> + n)
Temperate	103.81	54	137	11.68	0.04377565	O(n <sup>2</sup> + n)
Temperate -> Swap	101.54	52	133	11.8	0.058823529	O(n <sup>2</sup> + n)
Temperate -> 2-Opt	<b>88.75</b>	<b>47</b>	107	7.54	0.17373461	O(n <sup>2</sup> + n)
Temperate -> Swap -> 2-Opt	89.00	<b>47</b>	107	7.09	0.177838577	O(n <sup>2</sup> + n)

Note these arrows represent sequential executions of the different algorithms. In which the next one takes the optimized route from the previous.

The green boxes are used to highlight the best value for each metric.

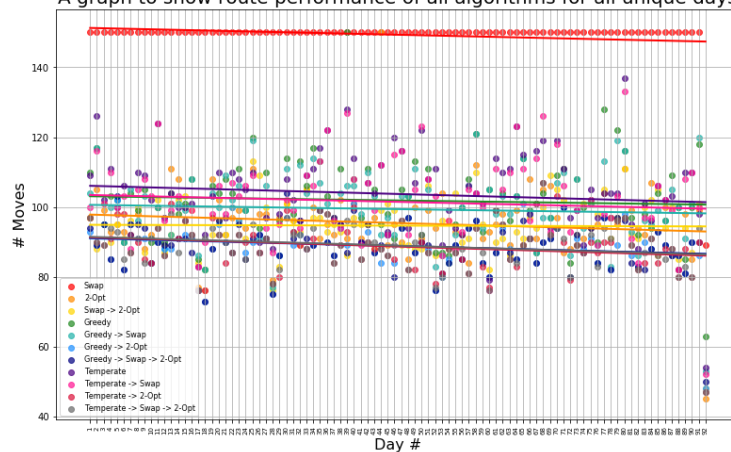
A graph to show the performance of all algorithms for all days in 2020 & 2021



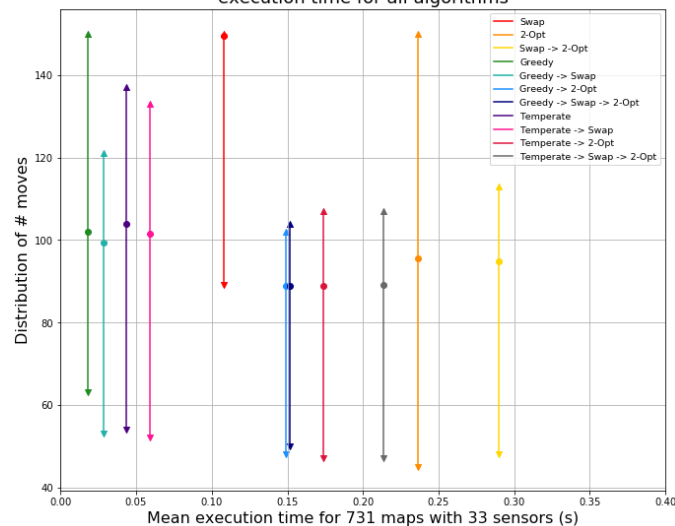
Upon testing I came to realize that the same sensors were used every 92 days. This is evident by the repetitive pattern shown in the plot on the right which shows the performance of all the different algorithms for all the days.

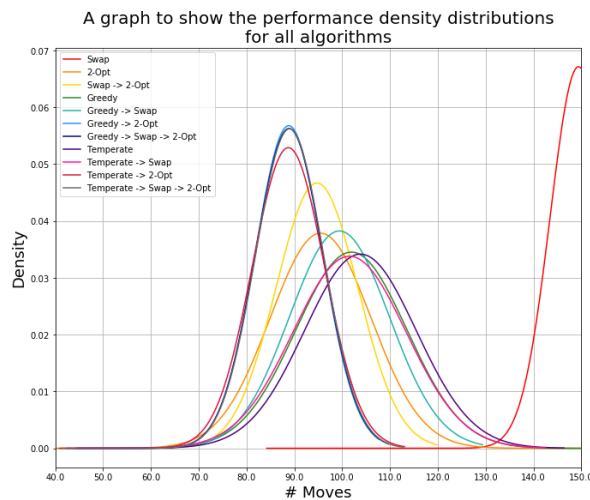
Therefore, for the sake of clarity, when plotting scatter plots to show the performance for each individual day I will only include these unique 92 days.

A graph to show route performance of all algorithms for all unique days



A graph to show the relationship between performance and execution time for all algorithms





### Optimal algorithm for performance, execution time and scalability

I am very happy with the results of my algorithms as they provide many effective solutions with varying trade-offs.

Although, the Temperate -> Swap -> 2-Opt option provides the best average number of moves, in the context of scalability this algorithm is less favourable given its high worst-case time complexity. This would be problematic when mapping a route between sensors across the

whole of Edinburgh. Given the area of Edinburgh is roughly 264km<sup>2</sup> and that the density of sensors would be the same as that for our current confinement area we can estimate there would be almost 30000 sensors needed to be visited. This is evidently a massive jump from 33 and would exponentially increase the execution time of our algorithm. So, choosing an algorithm with the right balance of performance and time complexity is extremely important.

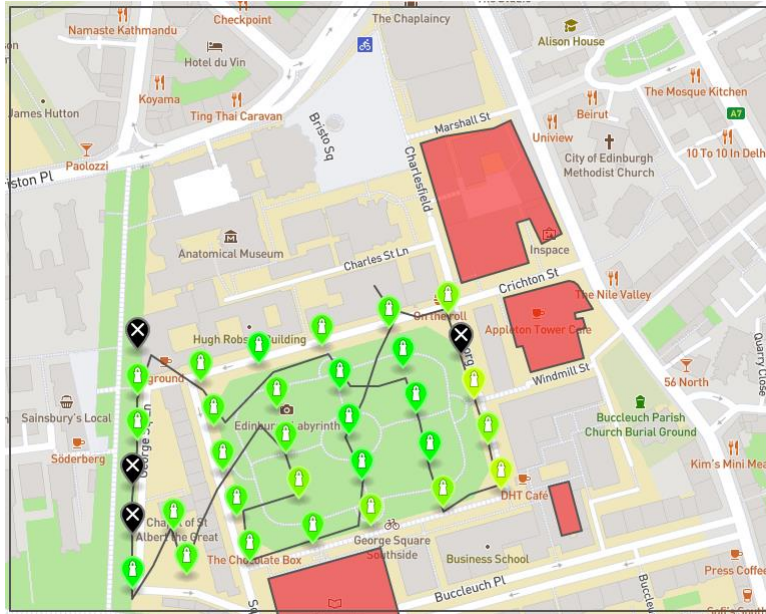
**\*\*An efficient algorithm can end up being faster due to it having to calculate less moves**

Furthermore, I believe consistency is very important when choosing an algorithm as it allows for more predictable results. This consistency is measured by the standard deviation of performance, the lower the standard deviation the more consistent the results are.

Given these factors, I believe in this context the Greedy -> 2-Opt algorithm would be the most effective to use. This is evident as it has the lowest standard deviation (excluding Swap given this result is due to poor performance), the best minimum number of moves achieved, the best maximum number of moves achieved, is only 0.45 moves off the best average, and has the best worst-case time complexity for all the algorithms with sub 100 move averages.

## Map outputs

Best # moves map (01-04-2020): 48



We can attribute this extremely low move count to be due to the lack of complexity in this example.

This is evident by the particularly close proximity of all the sensors and lack of buildings in the way.

Worst # moves map (05-01-2020): 102

