# Informatics Large Practical Project : Autonomous Drone Route Design with A* Search

**Jiaqing Xie**
UUN: 2001696
University of Edinburgh

## Contents

# 1 Project introduction

## 1.1 Background

Autonomous drone flying is popular nowadays. It can take pictures or record data at the given point or area by GOPRO or other devices. Designing the route with least steps and least time is essential to help researchers analyze data efficiently. According to this project, we are asked to design an algorithm, which aims to move drone through all sensors in the range of 0.0002(unit in degrees) and back to the point which is near to the initial point within 150 limited steps. It should also avoid flying inside the none-fly zone or outside confinement area at the same time. If we are successful in managing the route, it will provide great help for further researches and bring benefits to future projects. Project details can be found in specification file [1].

## 1.2 Data

We have been given three kinds of data files. First one is the no-fly-zone data file . No-fly-zone data presented in the file `no-fly-zone.geojson` describes the area that drone is not allowed to fly across. We can detour by choosing different angles to avoid entering these areas. Second kind of data is the data that records the information of a sensor's location, battery and reading, which are listed in the maps folder. Under one specific date in the maps folder has one `air-quality-data.json` file. The last kind of data is the data that records a what3words location's longitude and latitude , which are listed in the words folder. One location's concrete information is in the details.json file. Useful information are longitudes and latitudes.

## 1.3 Structure introduction

In this project, the documentation file is being given to illustrate how the classes are being created and what aims do they serve to. Also, the pictures will explain some of the functions in order to let the readers, users and researchers understand better on the functions, classes and algorithms. The algorithm is explained in part 4. Main graph search algorithm that we choose for this project is A* search algorithm. We've modified somewhere to the original A* search algorithm to serve for this project.

Software hierarchy and structure is shown in part2. Also, the given api is shown for users to use. Functions will be explained in part3 and are shown in the UML image. Open source codes will be uploaded on Github by permission.

# 2 Software architecture description

Totally eight classes have been created in the whole program, including the main class APP. The rest classes are `ParseURL`, `WordDetails`, `Record`, `BlockArea`, `Display`, `Node` and `FileGenerator`. One base class is `ParseURL`, which will be further used by `BlockArea` and `App`. This class is used for reading files from the given file's url(url parsing), which is essential since we must use url address to read files instead of passing filenames to the command line. `BlockArea` is created for parsing no-fly-zone.geojson file and create polygon features to reserve that information. Without this class, executing A* search algorithm is of no sense, which will be equivalent to greedy algorithm with no obstacles and also be failed in the tests for APP. The classes that use `BlockArea` is `Node` and `App`. `Node` is the dominant class in this program. It executes the main A* algorithm and describes the information of one node in the map(longitude, latitude, parent and gscore mainly). App class also depends on `Record`, which parses the json file into key-value pairs. We will definitely take use of the sensor's locations to plan route design. They are considered as the input for our A* algorithm.

App class includes the class `WordDetails` that maps a what3words location longitude and latitude. `FileGenerator` is used to generate flight path files and output geojson files. Each line in the flight path file must be int,double,double,int,double,double,string. And in the geojson file must include point features and line string features, which is depended on the class `Display`, which aims to add specific point feature properties with given colors and marker symbols.

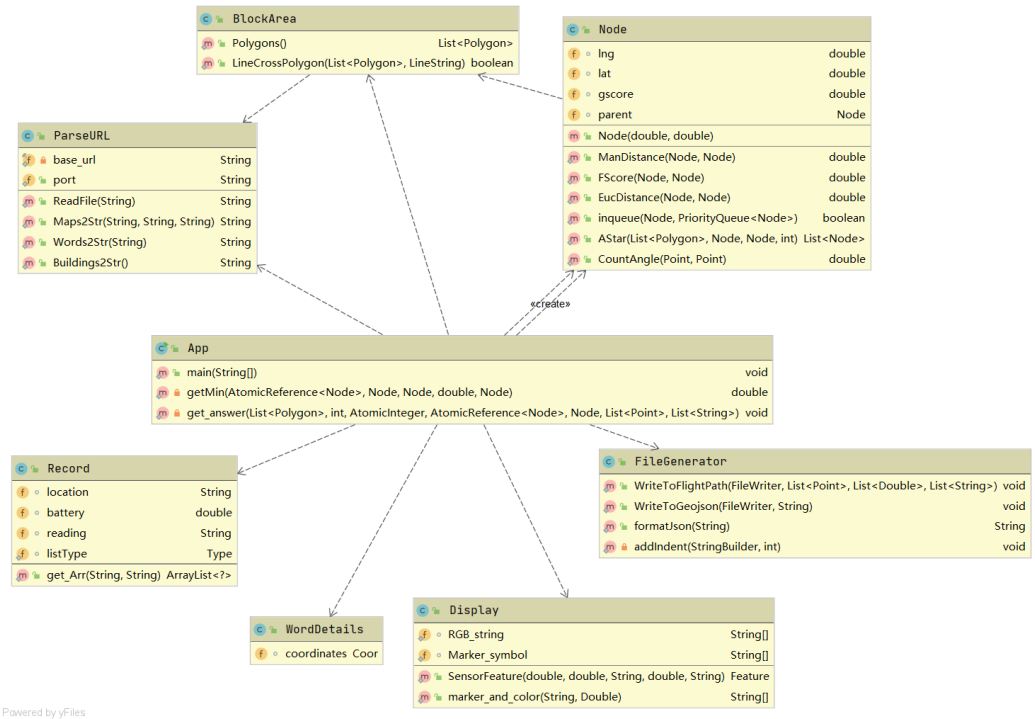



Figure 1: Overall structure represented in the style of UML

Figure 1 shows the structure of overall program. Each class represents its members including constants and functions for use(API). `Node` is the core of the program. Figure 2 shows a hierarchical structure of the program, which shows the hierarchical relationship between each class.

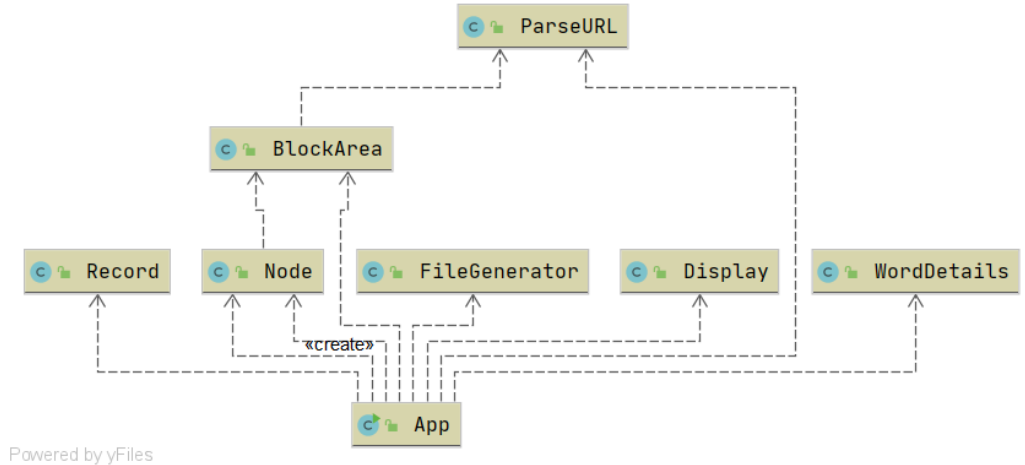



Figure 2: Hierarchical Structure

# 3 Class documentation

## 3.1 Class : ParseURL

### 3.1.1 Aim : Parse url address

In this project, the file described in json format should be accessed by running WebserverLite.rar on a specific port. ParseURL is the class that is designed for parsing files in maps, buildings and words folders when connecting with the specific localhost. Two class member variables are **base_url** and **port**, which are of private type and public type separately. They are defined as shown below. Since port is the parameter that we can pass to the command line, we set it as the public variable. Default port for this Webserver is 80. If we change a different port to the webserver, we must also change the corresponding parameter in the command line passing to the main class. The prefix "http://localhost:" can not be changed in main Java Class, which is **App** in this project. Therefore, base_url is set to private and final type.

Besides, filename in one folder under maps is the same as another one under the same main folder, which is air-quality-data.json, we set it as the constant parameter. They are of private and final type. It's similar in the words folder, where all files are named details.json. We set another priavte and final parameter called words_name. There is only one geojson file in the buildings folder, which is named as no-fly-zones.geojson. We also set a private and final parameter to reserve the filename.

```
private static final String base_url = "http://localhost:";
public static String port = "80";
private static final String maps_name = "air-quality-data.json";
private static final String words_name = "details.json";
private static final String buildings_name = "no-fly-zones.geojson";
```

Four member functions are defined as

```
String Maps2Str(String dd, String mm, String yy);
String Words2Str(String s);
String Buildings2Str();
String ReadFile(String s);
```

### 3.1.2 Function Details

Function Maps2Str has three parameters, which are dd(day), mm(month) and yy(year). They are all of integer type. This function maps the the air-quality-data.json filename to its url address with given day, month and year. Finally it returns the string representation of a given url address. For example, if the command parameters for these three parameters are 02 02 2021, then it will return the string http://localhost:80/maps/2021/02/02/air-quality-data.json

Likewise, function Words2Str maps a details.json filename to its url address at a cetain what3words address, which also returns the string representation of that url address. Function Buildings2Str maps a geojson file with none-fly zones to its url address with a similar "return" aim. The returned string can be further parsed by the function ReadFile. Given a url address string, this function will call a http client and ask for a http request.[2] The client sends the request so we will get the response body of that, which are the final json strings from the original file. In ordered to parse json strings, we have written class WordDetails and class Record to realize the purpose of that.

## 3.2 Class : WordDetails

### 3.2.1 Aim : Reserve what3words location coordinates

There are two ways to record the longitude and latitude of a what3words location. One way is to directly use what3words API, but it requires a lot of transformation works and also is not necessary. Therefore, in this project, the class that describes the information of a what3words location is created, which is named as WordDetails. It's also named as a class constructor. The class includes an internal class called **Coor**, which has two double type members named **lng** and **lat**. Variable **lng** is

used for recording the longitude while **lat** is used for recording the latitude. Other keys in json file, including **country**, **square**, **nearestPlace**, **language**, **words** and **map** are not used. Therefore we do not include them in the code files. Researchers can add them to the WordDetails.java if they'd like to present or utilize these details. The structure of this class is shown below.

```
public class WordDetails {
    Corr coordinates;

    public static class Corr {
        double lng;
        double lat;
    }
}
```

We can realize this method in the main class : `APP`.

```
var details = new Gson().fromJson(ParseURL.ReadFile(ParseURL.Words2Str((String)
    location.get(i))), WordDetails.class);
```

Here Gson().fromJson() is one method carried from com.google.gson.Gson, which has the ability to map a key-value pair json string to a given class object. "details.coordinates.lng" operation can achieve the longitude while "details.coordinates.lat" operation can get the latitude.

### 3.3 Class : `Record`

#### 3.3.1 Aim : Make key-value pairs on json files

There are 33 sensors' information recorded of each day in year 2020 or 2021 in the **maps** folder. We'd like to parse the json file under this folder and get json key-value pair as it has done in Class `WordDetails`. Therefore, the Class : `Record` is created for this purpose, which is also a class constructor. `Record` has three class members, which are **location**, **battery** and **reading**. Each of them can reserve the values from the parsed json files. **location** is of string type, **battery** is of double type and **reading** is of string type.

```
String location;
double battery;
String reading;
```

Arraylist is the prototype for us to reserve arrays of data. Here, three arrays should be returned. They are the arrays that reserve 33 what3words location names (string), 33 battery values (double) and 33 reading values (string). Therefore, we create the method `get_Arr(String s, String k)` to satisfy our need, where s is the jsonstring and k is one of the key words. Key words are **location**, **battery** and **reading**.

#### 3.3.2 Function Details

As it has shown in part 3.2, we first complete the mapping process by applying the `Gson().fromJson()` method[3]. We will get the **recordlist** that reserves 33 json key-value pairs for each day. We initialize two arrays: `string_arr` (String type) and `double_arr` (Double type) in order to return them at the end. If key word is equal to **location**, then firstly from **recordlist** get its sensor number i in order, then get its member **location**, which is realized by `recordlist.get(i).location`. The operation is similar when key word is equal to **battery** and **reading**. At the end, if key word is equal to **location** or **reading**, return `string_arr`, else return `double_arr` . One important issue is that battery array should be an array of double type while reading array should be an array of string type. Reading list might include values like "NaN" or "null", which is not able to be changed to a double number.

### 3.4 Class : `BlockArea`

#### 3.4.1 Aim : Judge if fly across the no-fly zone

In this project, autonomous drone cannot fly across a none-fly area, where the areas are the Appleton Tower, the David Hume Tower, the Informatics Forum and the Main Library. The information about any of a none-fly area is recorded in the `no-fly-zones.geojson` file in the buildings folder. Features are of the point features that describe the building area. The idea to create the class `BlockArea` is to judge if any of the polygon will have the intersection with our route. This is useful to avoid many route choice since there are 36 directions in our total choices. Sometimes some of them are meaningless because the next point will let them fly into a no-fly area. We have two functions in this class, which are `List<Polygon> Polygons()` and `boolean LineCrossPolygon(List<Polygon> p, LineString ls)`.

We use two pictures to illustrate the situation that the route is illegal. For example, in figure 3, the line string consists of two points, which has two intersection points with the no-fly-zone. In this case, the boolean function `boolean LineCrossPolygon(List<Polygon> p, LineString ls)` returns the answer false. Else the answer will return true. In figure 4, the line string consists of two points, one of which is outside the confinement area. The function will also return false under this case.

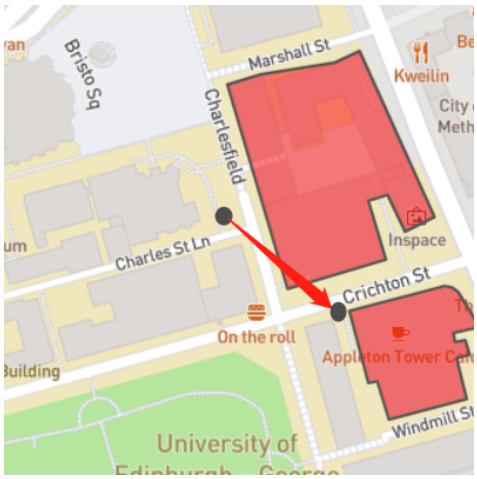



Figure 3: No-fly-zone

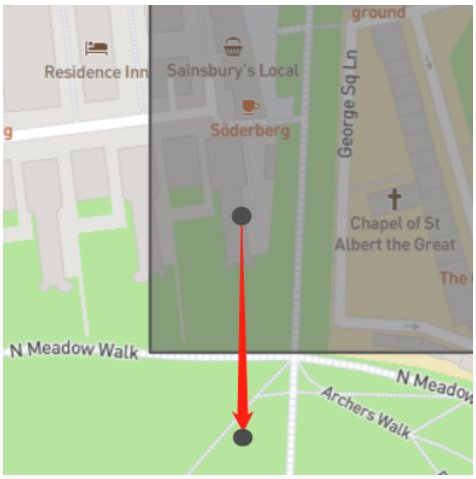



Figure 4: Outside confinement area

#### 3.4.2 Function Details

`List<Polygon> Polygons()` is used for reserve a list of **Polygon** objects, which map to each non-fly polygon area. We get feature collection from the json string **ba**, where ba is the parsed geojson file string.[4] Then we create the variable **f** to generate the list of features (important! not list of Polygons!). For each feature, we can get the geometry object and transform the geometry object to Polygon object, which is an allowed and legal forced conversion. The simple code that realize this function is shown below. After that we add each polygon object to the polygon list **pp**.

```
Geometry g = f.get(i).geometry();
pp.add((Polygon) g);
```

`boolean LineCrossPolygon(List<Polygon> p, LineString ls)` is used for judge if we will step into the no-fly zones. Parameters are list of polygons **p** and `LineString` object ls. The problem is equal to judge if total lines or edges of a polygon has the intersection with the input linestring ls. So here we set four coordinates for a line. The current longitude of a given point is set as `cur_lng` and the current latitude of a given point is set as `cur_lat`. The next point's longitude and latitude is set as `nxt_lng` and `nxt_lat`.

```
double cur_lng;
double cur_lat;
```

```
double nxt_lng;
double nxt_lat;
```

After that, we create a Line2D object with the coordinates that we have been given. This package is allowed to be used in this project. After creating the Line2D object `myline`, we use the method `intersectsLine()` to judge if the line has an intersection with each edges of a given polygon. In this project, there are four polygons. So the for loop will execute for four times.

```
Line2D myline = new Line2D.Double(c.get(0).longitude(),
c.get(0).latitude(),c.get(1).longitude(),c.get(1).latitude());

ans = myline.intersectsLine(cur_lng, cur_lat, nxt_lng, nxt_lat);
```

In this class, the polygon of confinement area is not added. Instead, we add it in the main class. The reason is that we want to directly utilize the information of the file that we parsed from no-fly-zone.geojson. Adding the feature of the confinement area requires the details of other class, which makes this class complex.

### 3.5 Class : `Display`

#### 3.5.1 Aim : Setting Custom Attributes

We have to add properties to the sensor points. These properties include geometric properties `coordinates`, `marker-symbol`, `rgb-string`, `marker-color` and `location`. The markers on the map have a symbol which is either a lighthouse, a skull-and-crossbones or a cross according to the value of readings in air-quality-data.json file. In this class, two private and final variables are fixed. They are `RGB_string` and `Marker_symbol`. The size of `RGB_string` is equal to 9. Values are `"#00ff00", "#40ff00", "#80ff00", "#c0ff00", "#ffc000", "#ff8000", "#ff4000", "#ff0000", "#000000"`. The size of `Marker_symbol` is equal to 3. Values are `"lighthouse", "danger", "cross"`.

```
private static final String[] RGB_string = {"#00ff00", "#40ff00", "#80ff00",
    "#c0ff00", "#ffc000", "#ff8000", "#ff4000", "#ff0000", "#000000"};
private static final String[] Marker_symbol = {"lighthouse", "danger", "cross"};
```

The reason that we use final values is that it is easy and convenient to read from the array. The repetition of local variables are redundant.

#### 3.5.2 Function Details

To realize this class, we should import two classes from the packages firstly.

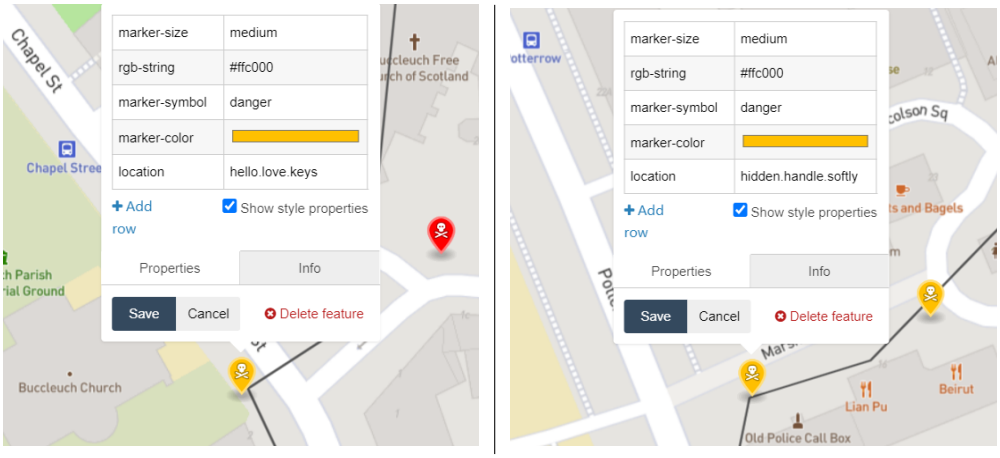
```
import com.mapbox.geojson.Feature;
import com.mapbox.geojson.Point;
```

Function `Feature SensorFeature(double lng, double lat, String Reading,double battery, String location)` is used for adding point features, which can be shown on geojson.io website. The most important thing is to judge if the reading (string type) is equal to "**NaN**" or "**null**" or if battery is less than 10. If so, marker symbol is equal to "**cross**". rgb-string is equal to "#000000"(black). marker-color is equal to rgb-string, which is "#000000". Under the condition of a less -than-10 battery value, no matter the value is null or NaN, it's not readable and unbelievable. Even if the battery value is larger than 10, the value can even be NaN or null. Therefore we set the judging conditions as mentioned before.

If reading is not equal to "**NaN**" or "**null**", it means that reading value can be changed to a double type value. We create another function to return a string array which includes the values for marker-color and marker-symbol. In a certain range, marker-color and marker-symbol is equal to a corresponding value.

For example, At the date 2020/01/01, a what3words location **hello.love.keys** has the reading 133.21. Therefore, it has a marker-color property of gold color, as well as a marker symbol of **danger**. At the

date 2020/02/02, a what3words location **hidden.handle.softly** has the reading 154.48. Therefore, it has a marker-color property of gold color, as well as a marker symbol of **danger**.




### 3.6 Class : `Node`

#### 3.6.1 Aims

We implement A* graph search algorithm in this project. To reserve the parent node, the longitude and latitude of the node and the gscore of the node, the Class `Node` is created. The class has one constructor, therefore the node can be initalized as `Node node = new Node(a,b)`, where a is the longitude of the node and b is the latitude of the node. It also has the parent attribute, which is used to trace back a valid route by finding its father node iteratively until its father node is equal to null or nothing.

```
public Node(double lng, double lat) {
       this.lng = lng;
       this.lat = lat;
    }
double lng;
double lat;
double gscore;
Node parent;
```

In this class, there are useful and powerful functions to calculate the distance between each two nodes. The distance metrics that we choose for A* search are Manhattan Distance and Euclidean Distance. More specifically, functions are named as `EucDistance` and `ManDistance`, where parameters are two nodes of Node type. Besides, there is a boolean function to judge whether the node is in the priority queue. This is helpful in A* algorithms, which will be detailed in part 3.6.2.

In the algorithm, it's difficult to record the angle of each two nodes when applying the A* search algorithm. In order to simplify the angle reservation preocess, we'd like to perform it when whole A* seach algorithm reaches its end. Such function is called `CountAngle(Point a,Point b)` , which can easily return the angles of a given starting point and a ending point. We certify that the algorithm will generate the right answer.

The most important function in this class and even in the while program is `AStar`. It aims to solve the graph searching problem in the least times with a more accurate route. It's more detailed in part 3.6.2.

#### 3.6.2 Function Details

Function `ManDistance` has two parameters, which are Node a and Node b. The distance is calculated by: $d = |a.lat - b.lat| + |a.lng - b.lng|$. Function `EucDistance` has two parameters, which are Node a and Node b. The distance is calculated by: $d = (|a.lat - b.lat|^2 + |a.lng - b.lng|^2)^{\frac{1}{2}}$. This is what they are defined in mathematics. The boolean function `inqueue` has two parameters, which

are Node a and PriorityQueue<Node> b. During the for loop, if a has the same longitude and latitude as one of the node in b, then return true, which means that Node a is inside priority queue b.

According to the angle counting function `CountAngle`, it also owns two parameters, which are point a and point b. To understand the function more intuitively, we can set starting point a to the origin (0,0) on a two-dimension x-y plane. Then ending point b must lay inside the Quadrant I or Quadrant II or Quadrant III or Quadrant IV, or on the x or y axis. In each area has a different angle calculation for the vector. You can see them in the code files. One important thing is that the angle ranges from 0 to 350, which are a set of non-negative mulitples of 10. So if the original angle is negative and the ending point is in the area of Quadrant IV, add 360 to the angle. If the ending point is in the area of Quadrant II, add 180 to the angle. If the ending point is in the area of Quadrant III, the angle is positive, but we also need to add 180 to the original angle.

Function `AStar` is the most important in this class. We just tell how to realize our rectified A* search algorithm here. The detailed algorithm will be discussed in part 4. We have four parameters for this function, which are list of polygons **p**, current node **cur**, next node **nxt** and maximum openlist_size of int type.

It's important to compare the distance penalty of each node. In A* algorithm, the function f(node) = g(node) + h(node) is set to compare the performance of each node, where g(node) is the penalty from the starting point to this node and h(node) is the penalty from this node to the ending point, which is also called the heuristic function. If the node has the total least penalty, it will be chosen to the next point on the route. Therefore here we set a lambda expression in Java : `Comparator<Node>` to be passed to the PriorityQueue, which is easy to pop the point node with the least penalties. Here, in this project, the heuristic function h(node) that we choose is the euclidean distance from the current node to the ending node. We set an openlist and closelist to reserve nodes that are not in our answer and the nodes that are in our answer.

Let's begin the algorithm. If the cur and nxt node has a distance less than 0.0002, which means that they are close, we just find the point which is next to the nxt node. Also, they should not be in the no-fly zone. The judging functions are:

```
Node.EucDistance(cur, nxt) < 0.0002; //distance smaller than 0.0002
if (!BlockArea.LineCrossPolygon(p, lineString)) {
    openlist.add(new_node);
}
```

. If they are not in no-fly zone, we can add them to the openlist. After that, we pop the first element, which is the optimal node that we should visit and add it to our answer list, we should also add our starting point to the answer list, and return the anslist.

```
anslist.add(openlist.poll());
anslist.add(cur);
```

However, unfortunately, if the cur and nxt node has a distance more than 0.0002, we perform the A* main algorithm. We poll the first element from the priority queue openlist. If it has a distance less than 0.0002 with the ending point, when the A* algorithm reaches its end, we add the current point, its father iteratively to the answer list until the cur node. Else, add n to the closelist, generate the 36 nodes with degree-fixed directions. If they are in closelist, then continue. If they are not in closelist and in openlist, add all nodes that are not in no-fly zones and outside confinement area to the openlist for next round check. If they are in openlist, just compare if it appears before. If not appeared, then it's not matter and just continue. Else we should compare the current node's gscore with the past node's gscore(history record). If the new node's gscore is smaller, then substitute with the new gscore and set the current parant to that node. If not, then continue and go to the beginning to judge if openlist is not empty. Algorithms ended if the aim sensor is visited in range or openlist is empty or openlist size reaches its maximum value.

### 3.7 Class : `FileGenerator`

We are asked to generate two types of files in this project. The text file includes current sum of steps, starting point(longitude and latitude), angle, ending point(longitude and latitude), the nearest location

in range(in the name of what3words location ). Geojson file has two features, one is for each sensor point and the other is for line strings. Therefore function `WriteToGeojson` is being created with two parameters file of type FileWriter and jsonStr of type String.

The output flight path file is strictly styled. The format of each line is int, double, double, int, double, double, string. To generate such file, the following code ensures the correctness of the style, as well as each path in each line, with correct coordinates and correct directions(0 -350 degrees).

```
pathfile.write(""+(i+1)+","+clan+","+clat+","+
Math.round(angles.get(i))+","+nlan+","+nlat+","+sensors.get(i));
```

In order to generate geojson file, we must get the filewriter. In the main class, this filewriter is being created if the total sum of steps is the optimal at current. Here, suppose we get the optimal sum at a specific loop. We have the jsonstring, which are from point features and line string features to its json format. There is an example of an output flight path file, which is of date 2020/06/06, with the initial point (-3.188396, 55.944425).

```
1,-3.188396,55.944425,350,-3.188100557674096,55.9443729055467,null
2,-3.188100557674096,55.9443729055467,0,-3.187800557674096,55.9443729055467,thank.salsa.brain
3,-3.187800557674096,55.9443729055467,100,-3.187852652127396,55.94466834787261,hurt.green.filer
4,-3.187852652127396,55.94466834787261,300,-3.187702652127396,55.94440854025147,null
5,-3.187702652127396,55.94440854025147,290,-3.1876000460843983,55.944126632465235,null
6,-3.1876000460843983,55.944126632465235,290,-3.1874974400414007,55.943844724678996,begins.spider.drips
7,-3.1874974400414007,55.943844724678996,180,-3.187797440041401,55.943844724678996,null
8,-3.187797440041401,55.943844724678996,180,-3.188097440041401,55.943844724678996,null
9,-3.188097440041401,55.943844724678996,180,-3.1883974400414012,55.943844724678996,null
10,-3.1883974400414012,55.943844724678996,170,-3.188692882367305,55.9438968191323,null
11,-3.188692882367305,55.9438968191323,170,-3.188988324693209,55.9439489135856,noted.friday.jams
12,-3.188988324693209,55.9439489135856,190,-3.1892837670191128,55.9438968191323,null
13,-3.1892837670191128,55.9438968191323,190,-3.1895792093450166,55.943844724678996,burn.spot.across
14,-3.1895792093450166,55.943844724678996,190,-3.1898746516709204,55.943792630225694,null
15,-3.1898746516709204,55.943792630225694,190,-3.1901700939968243,55.94374053577239,sculpture.shot.melon
16,-3.1901700939968243,55.94374053577239,260,-3.1902221884501243,55.943445093446485,arts.dish.scarcely
```

Figure 5: flightpath-06-06-2020.txt

In this class, we also define a function to format the json strings in order to read these geojson strings conveniently. If the string meets the character '{' or [', then add indent to the front, move the writing buffer to the next line . If it encounters '}' or ']', then delete an indent from the front of the line and move to the next line. When the character becomes ',', then just move to the next line and add nothing. If it meets the segmentation symbol ':' for keys and values, add one blank space before and after ':'. The indent adding function is realized by `addIndent`.

### 3.8 Main Class : `App`

App is the class that utilize all the other classes and functions to realize A* search in autonomous drone flying and generate outputs. We will introduce it by splitting into different parts.

The first part is about importing packages. We are told that we are not allowed to use payed or confidential packages. In this project, all the packages are public and not paid. They are `java.io`, `java.util`, `com.google.gson`, `com.mapboc.geojson` and `java.lang.Double`. They are for reading parameters from command line, writing files, constructing array lists and performing array operations, using concurrent process to filter long paths, using open source api to write geojson files and using internal methods to change a string into a double number.

The second part is about setting local and global variables. We take the parameters from the command line, and get the reading, location and battery arrays for the specific day that command line parameters have determined. We set the list of point lists `best_pointlist` to reserve the best point lists(in order). `pol` is used to reserve no-fly-zones. `Record_copy` is being refreshed for each openlist size. We create the feature list `featureList` to be transformed into feature collection finally. We add the sensor's feature properties to the feature list for each date. We have set a list of openlist maximum size to choose from. They are the multiples of 36 or multiples of 10.

The third part is that we are going to realize the A* algorithm for each openlist size maximum. We initialize the current node and next node, as well as three local variables `angles`, `Ans_point` and `Record_Node` to reserve current solution but are not optimal. The optimal answers will be reserved in global variables. At the beginning, if the starting point is in range(within the distance of 0.0002), then just perform one move and back to the initial point. Else perform A* directly until in range of 0.0002. After removing the first sensor, the program starts to perform A* iteratively until all sensors are visited. Finally if the last point and initial point are in range, then stop. Else continuing perform A* until they are in range. After that, we compare our answer to the optimal steps. If it becomes the best answer, then substitude the answer that we've got at the last round, write the elements in the arrays to text files and geojson files. Importantly, the calculation might not be ended in some seconds, which means that the openlist size is not appropriate. We use try-catch blocks and threads to ignore these openlist maximum values.

The fourth part is about the function that we repeatedly used in the App class, which are `getMin` and `get_answer`. `getMin` is the function to get the nearset sensor while moving each time. Values are changes through parameter passing. `get_answer` is used for get the sum of steps from the current node to the ending node(nearby), as well as adding points to the point list and adding each move's information, which is null if not reaching ending point or the what3words location of the ending point if reaching the end.

# 4 Drone control algorithm

Traditionally, graph search algorithms include depth-first search(`DFS`), breadth-first search(`BFS`), random walk, greedy algorithm, Dijkstra algorithm and A* algorithm. In this project, since there are at most 36 directions for us to choose for each step, the time complexity and space complexity is really high if we want to represent all the possible combination of flight paths. That is $\mathcal{O}(36^n)$. For many algorithms like BFS and DFS, it might encounter memory overflow problems when encountering obstacles.

## 4.1 Rectified A* Algorithm

A* algorithm is an extended algorithm by Dijkstra algorithm. Since Dijkstra only consider the cost from the current node to the former node. However, it doesn't care anything about the current node to the ending node. In the practical graphs with obstacles, the program may step into dead locks because it will always walk back and forth and never pass obstacles. Therefore, in this project, we take the A* search as our main algorithm to complete the task. Importantly, greedy algorithm is also implemented in this program but is trivial. It is used when we start to fly to the next sensor after we complete taking the record from the current sensor. The distance between current node and not-visited sensors is measured. We'd like to choose the sensor with the shortest distance to the nearest sensor.

The main idea for the A* algorithm is the heuristic function.Dijkstra algorithm with heuristic function is A* algorithm as well. The total cost function f(n) is the function that we want to minimize, which is defined by:

$$f(n) = g(n) + h(n) \tag{1}$$

, where g(n) is the cost from the current node to the starting node and heuristic function, named as h(n), is the cost from the current node to the ending node. In this project, we choose euclidean distance expression for h(n), and g(n) is equal to the length of steps(multiples of 0.0003) from the starting node to the current node. The definition of heuristic function is really important. Apart from euclidean distance, Manhattan distance is also a good try for heuristic function. Some researchers and workers also choose the bias of Manhattan distance and euclidean distance as its heuristics function. You can change them in the Node.java file if you want. The other important rectification is that we set the maximum length for the openlist length, which is also equal to the numbers of tested points to some extent. In the traditional A* search problem, the directions are only limited to 8, where all things can be listed in the style of a matrix. However, this project needs to consider 36 directions before each move. Which means that nearly every points on the map can be readed. This will throw out a out-of-memory error to the computer.

This part will introduce how A* algorithm works in this project. We need an openlist to reserve nodes that is ready to visit. This openlist is a priority queue in practical since we need to extract the smallest element. Taking the advantage of priority queue in Java is time-efficient. We also need a

closed list(also a priority queue) to inform that nodes are visited. The inputs of the algorithm are starting point and ending point with known coordinates. If the starting node and the ending node is within the range of 0.0002, we just perform the move with 36 directions for once. Many users may ask if it will move outside range after the first move. The answer is : it doesn't matter, since if the next node is outside range, the program will execute the part of the algorithm that satisfies the condition that the nearest two nodes are not in range. After moving nodes by 36 directions, there might exist nodes that are in no-fly-zones or outside confinement area. We need to judge if they are not in these areas, then add these nodes to the openlist. The first element in the priority queue will be extracted at the next round. Importantly, before we add new nodes to the openlist, we need to record then remove the first element first.

Here comes the condition that starting node and ending node is not in the range of 0.0002. The algorithm is that : first we add the starting node to openlist. If the openlist is not empty(it's always true in the first round since the openlist always has a starting node in the first round). We set the node that is the optimal node, which is called sp here. Set the node p as the current node. We poll the first element from the openlist, if the polled node has a smaller manhattan distance or euclidean distance, a higher gscore, then replace sp with that polled node. The reason is that we set the maximum length for openlist length here. This is the parameter that we pass to in the main program. We should get the current optimal node after the size of openlist reaches its maximum. We find if the current node is in range, then return anslist from the leaf node to its root node through reading its parent nodes on the path. If not, add the polled node to the close list. If openlist size reaches its maximum size, then taking the optimal node sp as the leaf node and add its parent nodes iteratively until it reaches the starting node. If the size of openlist is smaller than fixed maximum value, for each direction we calculate the information of its next node(new node). If the new node is in openlist, then pass, since they have been visited(in the optimal path). If it is not in openlist and closelist, which means that it's a new generated node, then add it to the openlist. This node should also locate outside no-fly-zone and in the confinement area as well. If it is in the openlist, we should compare the new node with the old node. Since h(n) is the same , we will choose the one with a lower g(n) in order to minimize f(n) for each node. Here, the algorithm is over. It promises to return a list of nodes from the last node that is in range to the sensor, to the starting node.

This algorithm should return a multiple of 10 angles and a distance of 0.0003 of each path. Although it's not the most optimal algorithm since we set the particular numbers for the openlist size, the performance is much better and quicker than simple DFS ,BFS and greedy algorithms. The next page shows the overall rectified A* algorithm in a table.

**Algorithm 1:** Rectified A* Algorithm

**Input:** List of Polygons p, starting node cur, ending node nxt, openlist size maximum
**Output:** List of nodes anslist

```
if EucDistance(cur, nxt) < 0.0002 then
    Node new_node;
    for i ← 0 to 35 do
        new_node.lat = cur.lat + F1(i);
        new_node.lng = cur.lng + F2(i);
        new_node.parent = cur;
        if not in illegal areas then
            openlist add new_node;
        end
    end
    extract the first node, add to anslist;
    return anslist;
end
openlist add cur;
sp ← cur;
while openlist not empty do
    n ← openlist.first;
    if n is optimal then
        sp ← n;
        if EucDistance(n, nxt) < 0.0002 then
            while n not equal to null do
                add n to anslist;
                n ← n.parent;
            end
            return anslist;
        end
        closelist add n;
        if openlist size > maximum then
            while sp not equal to null do
                add sp to anslist;
                sp ← sp.parent;
            end
            return anslist;
        end
        for i ← 0 to 35 do
            if new_node in closelist then
                continue;
            end
            if new_node not in openlist then
                if new_node not in illegal areas then
                    openlist add new_node;
                end
            end
            else if new_node in openlist then
                find corresponding old_node in openlist;
                if new_node.gscore < old_node.gscore then
                    old_node.gscore = new_node.gscore;
                    old_node.parent = new_node.parant;
                end
            end
        end
    end
end
```


Figure 6: Date : 2021/07/16


Figure 7: Date : 2021/12/25

### 4.2 Tests for Rectified A* Algorithm

We have included two test images in this document. The test dates are on July 16th, which is my birthday and on December 25th, which is the Christmas Day.

The starting point that we choose is (-3.188396, 55.944425). The minimum sum of steps on 2021/07/16 is 105. The minimum sum of steps on 2021/12/25 is 96.

## 5 Conclusions and future works

In this program, we have learned to build a program with geojson api and utilize different data structures and algorithms to complete a task. We have known the superiority of A* search among mainstream graph search algorithms. We have known the importance of using local variables or final or private type values. We have known how to generate the files correctly and in a easier way.

My future plan for this project will be: trying more advanced algorithms to compare the A* algorithm with these algorithms. Ideally, we will build a website to show the discrepancy between those algorithms. We will try other data structures like hash tables to substitute priority queues.

## References


[1] Paul Jackson Stephen Gilmore. Specification. https://www.learn.ed.ac.uk/webapps/blackboard/content/listContent.jsp?course_id=_80461_1&content_id=_4788239_1&mode=reset.

[2] Paul Jackson Stephen Gilmore. The java 11 http client. https://www.learn.ed.ac.uk/webapps/blackboard/content/listContent.jsp?course_id=_80461_1&content_id=_4788239_1&mode=reset.

[3] Paul Jackson Stephen Gilmore. Json parsing. https://www.learn.ed.ac.uk/webapps/blackboard/content/listContent.jsp?course_id=_80461_1&content_id=_4788239_1&mode=reset.

[4] Paul Jackson Stephen Gilmore. Geo-json parsing. https://www.learn.ed.ac.uk/webapps/blackboard/content/listContent.jsp?course_id=_80461_1&content_id=_4788239_1&mode=reset.