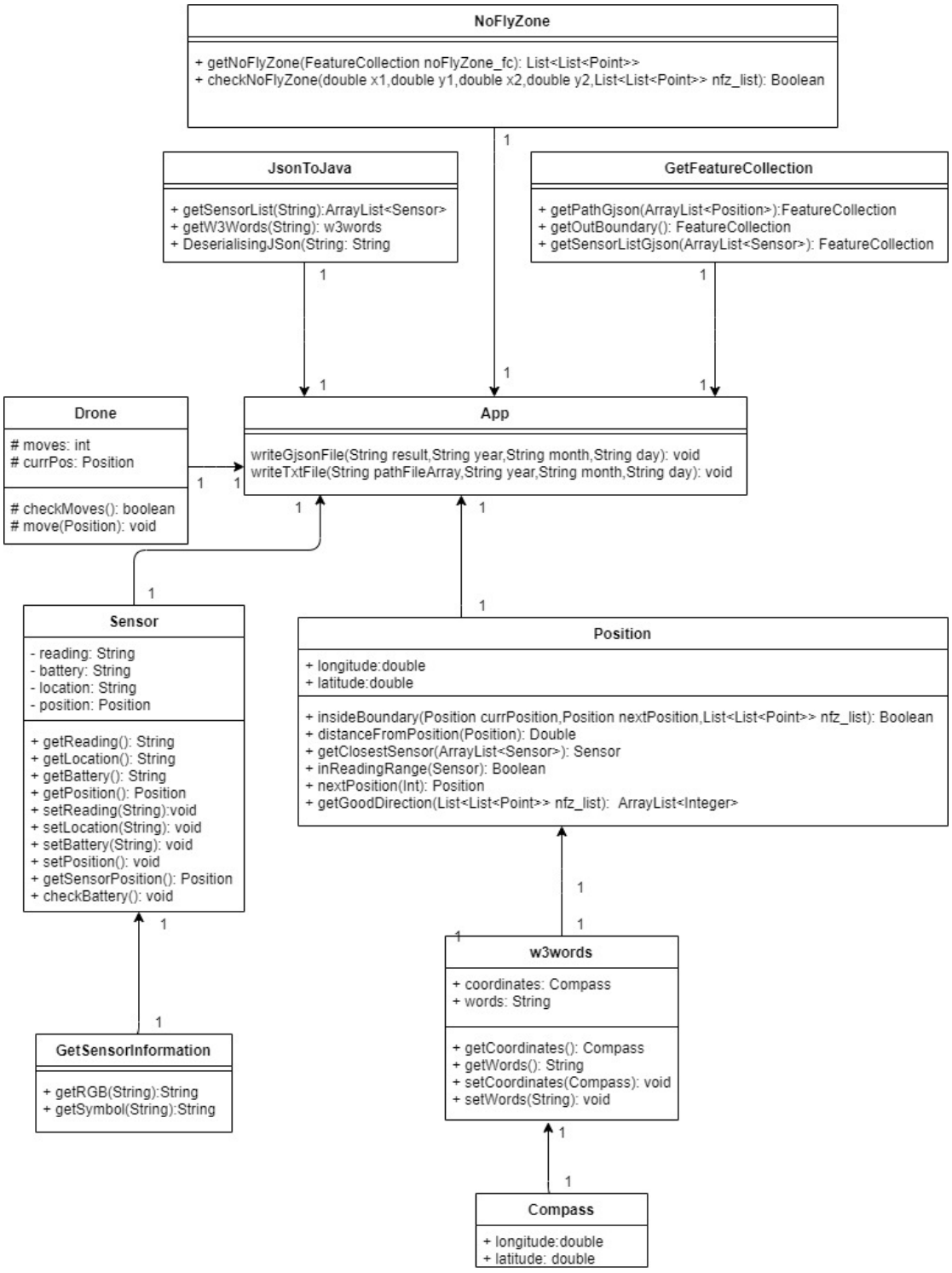
Informatics Large Practical
Coursework2
s1862323

1. Software architecture description

In this section, I will describe the software architecture of my application. The UML diagram in the next section will show the hierarchical relationship between the classes. A brief overview of the methods and variables in each class is provided.



1.2. Reasoning

The **App** class serves as the main entry point and launcher for the entire application. This is where input parameters are parsed and output files are generated. These variables are public, as they can be accessed from multiple external classes, such as the Position class.12/4/2020. It can set the sensor starting point. It has two method to generate the file we need.

The **JsonToJava** class, as a method class, has three static methods in order to keep the **App** class concise: it takes the formatted String as a parameter and outputs all the data we want. It converts the String from the Gjson style into java style data. By using various methods within the class (getSensorList, getW3Words, DeserialisingJSon). By the way, The **DeserialisingJSon()** method is used to deserialize the Gjson file into java style.

The **GetFeatureCollection** class also acts as a method class in order to keep the **App** class concise. It has three static methods that are used to generate different FeatureCollection from different parameters, which we can then easily use featureCollection to form a Gjson file. For example, using ArrayList of Position to form a Path FeatureCollection, and using ArrayList of Sensors to form 33 Sensors FeatureCollection.

The **Sensor** class represents a sensor object in the map, which has its own readings, battery, position and location attributes. All the attributes in the **Sensor** class are private, so there are getter and setter methods. Since the sensor's position is what3words style, the sensor's corresponding position is being reset by creating the methods setPosition() and getSensorPosition(). We can then use the getPosition() method to get the sensor's position directly instead of sending an http to get the value. In addition, the checkBattery() method is used to reset the battery value because of the coursework document required.

The **w3words** class is like a transfer station or a necessary complement to the **Sensor** class. It has two attributes (coordinates and words). Since in the Sensor class, the sensor's position is w3words style, we can get the sensor's accurate latitude and longitude by using w3words. Without this class, we would not be able to get the exact **Position** of the sensor. These variables are public because they are invoked in the Sensor class to generate the required output file.

The **GetSensorInformation** class is also a complement to the **Sensor** class in order to make the Sensor class concise. It has two methods for transferring the read value to an RGB String and a Symbol String. These variables are public, as they can be invoked in the **Sensor** class to generate the target variable.

The **Drone** class has two attributes: moves and Position. The **Drone** object has the protected variables moves and Position, which can be modified in the class. the value of moves is used to check whether the drone exceeds the maximum 150 moves. the move() method is used to control the drone's movements (to make the drone move to next Position). These variables are public, as

they are used in the app class to generate the required output variable.

The **NoFlyZone** class has two methods to ensure that the drone does not enter the no-fly zone. One is the getter method and the other is the check method to determine whether the drone and its next position will intersect the No-Fly-Zone. It returns a Boolean value to check if the drone's next position will intersect the boundary line, which if true means that the position is inside the no-fly zone and false means no intersect. These methods are static, as they can be invoked in the Position class to determine if a position is accessible.

The **Position** class has all the methods related to the Position object, such as determining if the drone is within the play area, finding the closest Sensor to the drone's current position, calculating the distance between drones current position and the another position, etc. The method getGoodDirection() takes a noflyzone list as parameter and returns all the direction which will not intersect the boundary and no-fly-zone. The methods in this class are used throughout the entire Drone class. And the Position object itself has the variables longitude and latitude, which are private final double variables that should not be changed during the entire runtime of the application.

1.3. Class Relationships

The **App** class is the launcher for the entire application. It gets the String year, month, day, latitude and longitude. All these data are taken from the arguments. Then the starting point of the drone is set through the latitude and longitude. The corresponding sensor list is then obtained by using the string that is a combination of "http://localhost:80/maps "+"/"+year+"/"+month+"/"+day+"/air-quality-data.json". This string can then be used to retrieve the list of sensors by using the methods in the **JsonToJava** class. In addition, all sensors are reset by applying the methods inside the **Sensor** class to form each complete sensor (because the sensor object is not complete when it was taken from the http by using **JsonToJava** class its methods). It has the location of the sensor (what3words style data but not java style data Position). Since the sensor class also will invokes the corresponding useful classes and methods (**w3words, GetSensorInformation**) to help it obtain the necessary values and information(the accurate Position of the sensor). Then it will invoke the **Position** class to find the closest sensor for the drone. In the **Position** class, it invoke the **NoFlyZone** class checkNoFlyZone() method to make sure that the drone is not in the no-fly zone. And then it uses an algorithm to get the best path for the drone. And using the nextPos () method to get the next Position of the drone want to go. if the drone is in the sensor reading area, it removes this sensor from the sensorList and allows the drone to move to the next sensor for detection. If all sensors are visited, then the drone will return to the starting point. Getting the pathfeatureCollection by using the methods in the **GetFeatureCollection** class. Finally, by using the Feature Collection which contains the path features, sensor features and the path features to form the final Feature Collection and also using the given data to form a String which contains the necessary information to write a txt file. In the end using the writeFile() method in the App class and take the Feature Collection String and the String to form two file: Gjson file and txt file.

2. Class Documentation

In this section I will discuss the classes in a more in-depth manner, specifying attributes within each class, the input and output of each method, their visibility and why they are integral to generating the movement of the drones.

2.1. App.java

This is the main controller class.

1. The **writeGjsonFile(String result, String year, String month, String day)** method takes the four String parameters, and then it creates the file with the fileName argument, and writes the String parameters to the file. This method generates the corresponding .geojson file, which is saved in the current directory.
2. The **writeTxtFile(String result,String year,String month,String day)** method takes the four String parameters, and then it creates the file with the fileName argument, and writes the String parameters to the file. This method generates the corresponding .txt file, which is saved in the current directory.
3. **The main(String[] args)** method of this class firstly reads the input parameters and then checks if the input parameters are valid, i.e., if the number of input parameters is correct. If the input is valid, then it will initialize the data we need.
4. The starting point is based on the argument longitude and latitude, then initialize it. The ArrayList<Position> path is used to store all the Position for drone to visit. The pathFileArray is used to store String, and finally to develop a .txt files. Then it invokes the JsonToJava class to get the SensorList and gets the corresponding Features by invoking method inside the GetFeatureCollection. And it invokes the JsonToJava.DeserialisingJSon() methods to get Json file and transfer it into Features by built-in method in the FeatureCollection. And the nfz_list is the variable(List<List<Point>>) to determine whether the drone will fly into no-fly-zone.
5. In fact, the while loop divides the drone into two cases, if the drone has finished reading all the sensors, then it needs to go back to the starting point. If not, then it continues to go to the sensor location to read as a priority target. For the greedy algorithm, the drone needs to search all the sensors and find out the closest one. And then the drone moves into it to read the sensor. Then after getting the closest sensor, we need to find the optimal path to access it. Then try its best to go the best direction to access the sensor. (The definition of my good directions is : will not intersect the no-fly-zone and make it as close to the target sensor as possible) Then, I created a list of Integer to store all the good direction in each drone current Position. And then, because sometimes the drone will keep moving between two points and get stuck on it. I also wrote down a way to remove some of the direction and position that have already visited. This method will largely decrease the drone get stuck. And then by getting the closest direction and position, the drone will move to this position by invoking the move() function inside the Drone class. When the drone accesses the sensor and reads the values, then this sensor need to be removed from the sensorList to avoid revisit. Moreover, all the information of each path will store on the path and pathFileArray, to make sure each step

are recorded and without mistakes. If all the sensors are read, the drone needs to go back to the starting point. Therefore, the drone will also have a series of good directions that do not intersect with the no-fly zone. Instead, choose the nearest direction and let the drone move to the starting point in fewer steps.

6. In the end, if all the sensors are read and the drone current position is on the 0.0002 degrees close to the starting point. Then, the Boolean value of end will switch to True. The whole while loop will terminate. And then it will invoke the corresponding methods to get the Features. Combining it all to get the whole feature collection and change it into String type. Finally, use Feature Collection String and the pathFileArray to get the .txt file and .gjson file by invoking the two write file methods.

2.2. Position.java

1. The **latitude** attribute which is of type double that represents the latitude of an object.
2. The **longitude** attribute which is of type double that represents the longitude of an object

Both attributes are public final as they should be immutable. The Position class also has the following methods

3. The **nextPosition(int index)** method takes int as a parameter and moves the drone from its current position to the next position when it moves in the specified direction. This is obtained by calculating the changes in longitude and latitude and adding them to the current longitude and latitude values. These new values are then used to create a new position object and the next position of the drone is returned.
4. The **insideBoundary(Position currPosition,Position nextPosition,List<List<Point>> nfz_list)** method takes the current Position and next Position, it returns a Boolean value for checking whether a location is inside the boundary, including the outer boundary and the No-Fly-Zone. If true then this position is accessible and without entering the No-Fly-Zone.
5. The **inReadingRange (Sensor sensor)** method takes a Sensor object as parameter and returns a boolean value to check whether the station is within the reading range of the drone, i.e. the distance between the drone and the sensor is less than or equal to 0.0002 degrees. If true then it means this sensor can be read.

6. The **distanceFromPosition(**Position nextPosition**)** method takes a Position as parameter and returns the Euclidean distance between the drone's current position and the position of the point. The return value is double. The Euclidean distance formula to calculate the distance between the drone's current position and the point position is the following. X is the latitude, and Y is the longitude.

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

7. The **getClosestSensor(ArrayList<Sensor> sensorList)** method takes the sensor's ArrayList as a parameter and returns the nearest sensor according to the current position of the drone. Comparing each sensor and calculating the distances between the drone and its

sensor. Getting the shortest distance and closest sensor

8. The **getGoodDirection(List<List<Point>>> nfz_list) method** takes List<List<Point>> as an argument and returns ArrayList<Integer>. Check if every 10 angles and its next Position is on the boundary by using 360 angles. If the Position is not on the boundary, then the angle and its position are not allowed.

2.3 Drone.java

1. The attribute **moves** which is of type int represents the number of moves for the drone.
2. The attribute **currPos** which is of type Position represents the current Position of the drone.
3. The **void move(Position nextPos)** method takes the Position as parameters. This method is used for moving the drone into next Position. And the number of moves will increase one.
4. The **checkMoves()** method takes no parameter and return a boolean value as output. This method is used to check the number of times the drone has been moved. If the number of moves exceeds 150, it returns false, otherwise it returns true.

2.4 GetFeatureCollection.java

The **GetFeatureCollection** class also acts as a method class in order to keep the app class concise.

1. The **getPathGjson(ArrayList<Position> path)** method takes ArrayList<Position> as an parameter and returns its featureCollection as an output. Since all Positions are composed of latitude and longitude, they can be used to generate a Point in mapbox.jeojson, so it can form a LineString, which then forms its features. Finally, a feature collection is obtained by combining the features. The feature set is very useful and convenient for getting Gjson files.
2. The **getOutBoundary()** method takes no parameter and return a FeatureCollection of the big external boundary of the playing area as an output. Since all the Point are taken from the given data, and then it can form a LineString, which then forms its features. Finally, a feature collection is obtained by combining the features. The feature set is very useful and convenient for getting Gjson files.
3. The **getSensorListGjson(ArrayList<Sensor> sensorList)** takes ArrayList<Sensor> as parameter and returns the Feature Collection of this sensorList. Because each Sensor is the element of the ArrayList, then we can have each individual Sensor. The data of each Sensor, such as readings, battery, position, location, etc., is obtained through the getter method inside the Sensor class. Hence, we can obtain a Point and its Feature by adding all the necessary information required. Moreover, a list of Sensor can used to form a list of Features, which combined together to get a Big FeatureCollection which is the Feature Collection of this whole sensorList.

2.5 GetSensorInformation.java

The **GetSensorInformation** class is also a complement to the Sensor class in order to make the Sensor class concise.

1. The **getRGB(String str)** method takes the reading string as a parameter and returns a corresponding RGB String. It takes the input string into double value, and return different RGB value because of its double value.
2. The **getSymbol(String str)** method takes the reading string as a parameter and returns a corresponding Symbol string. It takes the input string into double value, and return different symbol String because of its double value.

2.6 JsonToJava.java

The **JsonToJava** class, as a method class, has three static methods in order to keep the app class concise.

1. The **getSensorList(String str)** method takes a String as parameter and returns its corresponding data type. In this method, the string is a Sensor List in Gjson style. Therefore, this method is used to transfer the Gjson style String into Java style data. It is useful and convenient for the App class to invoke.

2. The **getW3Words(String str)** method takes a String as parameter and returns its corresponding data type. In this method, the string is a what3words Gjson style string. Therefore, this method is used to transfer the Gjon style String into Java style data. It is useful and convenient for the Sensor class to invoke.

3. The **DeserialisingJSon(String str)** method takes a String as parameter and returns its corresponding data type. This method is used to convert Gjson file into Java file by reading Json from URL. This method will send the http request to get the data we need. It is useful and convenient for the others class to invoke.

2.7 NoFlyZone.java

1. The **getNoFlyZone(FeatureCollection noFlyZone_fc)** takes the noFlyZone_fc feature collection as a parameter and return List<List<Point>> nfz_list. It is a complement method for the checkNoFlyZone()method.

The checkNoFlyZone(double x1,double y1,douole x2, double y2, List<List<Point>> nfz_list) methods take the four double value and a List<list<Point>> as the parameters and return a Boolean value to illustrate whether the drone intersect the no-fly-zone. if true means the next Position of the drone intersect the no-fly-zone. If false the it means it is not in the no-fly zone. The method is based on the line2D.linesIntersect

2.8 w3words.java

1. The Attribute **coordinates** which is of type **Compass** represents the Compass for the sensor. In the **Compass** class, it contains two attributes lat and lng, which are represent the latitude and longitude of the sensor. The data type is double for lat and lng. Hence, by using this class we can use the what3words data to get the accurate Position of the sensor (latitude and longitude).
2. The Attribute **words** which is of type String represent the what3words String for the sensor. It is a backup attribute so just ignore it.

2.9 Sensor.java

1. The Attribute reading which is of type String represents the Sensor air quality reading.
2. The Attribute battery which is of type String represents the Sensor battery value.
3. The Attribute location which is of type String represents the Sensor what3words location.
4. The Attribute position which is of type Position represents the Sensor Position.

All the attributes in the **Sensor** class are private, so there are getter and setter methods. Since the sensor's position is what3words style, the sensor's corresponding position is reset by creating the methods setPosition() and getSensorPosition(). We can then use the getPosition() method to get the sensor's position directly instead of sending an http to get the value. In addition, the checkBattery() method is used to reset the battery value because of the coursework document required.

1. The **setReading(String reading)** takes reading a String as the parameter and return nothing. This is the setter method. Because the air-quality value will not accurate when the battery value is low or null. Hence, we need to redefine the value of reading to make sure the reading is correct.
2. The **getLocation()** method takes nothing and return the location of the sensor. This is the getter method. It is public method to make it easy to invoke.
3. The **getReading()** method takes nothing and return the air-quality reading of the sensor. This is the getter method. It is public method to make it easy to invoke.
4. The **getBattery()** method takes nothing and return the value of battery of the sensor. This is the getter method. It is public method to make it easy to invoke.
5. The **getSensorPosition()** method takes nothing and returns the **Position** of the sensor. The **getSensorPosition()** method and the **setPosition()** method are a combination. Because the sensor only has three attributes reading, location and battery when the http requested and get the data back. We only has the what3words location of the sensor. Then we need to invoke the **w3words** class and **JsonToJava** class and the **DeserialisingJson(String)** and **getW3Words(String)** methods to get the accurate location of the sensor(latitude and longitude). And then we need to set the Position of the sensor instead of null. Because the setter and getter method are actually a combination. I produce another method getPosition() to get the Position of the sensor after the setting. Then it will increase the efficiency of the program because it doesn't need to keep sending requests to http and URL to get the sensor's position.
6. The **checkBattery()** method takes nothing and return nothing. Because it is a kind of Preventive Measures for the sensor. When the battery value is low or null, the reading of the air quality is unreliable. Hence this method is used to check the battery value and then reset the reading of the sensor under some conditions.

3. Drone Control Algorithm && Strategy

3.1 Overview

I use a greedy algorithm in the drone movement process. This algorithm and strategy covers how to get the nearest sensor, how to get the best next position, how to avoid the no-fly zone, and how to solve some problems in the movement: for example, in some cases, the drone will keep moving between two points until it runs out of moves. In addition, I will also mention the shortcomings and advantages of this algorithm. In addition, I have inserted two images on the next page to show the actual situation.

3.2 Algorithms brief introduction

At first, the App will initialize all the necessary data and store it in different variable.

1. **Finding the closest Sensor and set this sensor as the target sensor to move toward.** Get the nearest Sensor by using the current Position of the drone and the getClosestSensor() method
2. **Form a ArrayList<Integer>,** which each int represents a direction that the drone will move. And based on these integer, the drone can get the next Position due to a different int by invoking the nextPosition() method. The ArrayList<Integer> goodDirection, which means these direction will not intersect any no-fly-zone and boundary due to the current Position of the drone and its next Position.
3. **Find the best direction in which the drone will move and its next position.** It needs to get an optimal direction and its Position as the direction of the drone's movement. By calculating the distances between each next Position obtained by the direction and the target Sensor (closest Sensor). The app class can finding out the shortest distance and get the corresponding optimal next Position.
4. **Invoke drone move() method to move the drone and update the value of drone moves**.
5. **keep moving it until the current Position of the drone is under the reading Zone** (the area that the drone can read the sensor value).
6. **ArrayList remove the closest sensor,** which to avoid the drone keep moving around the same sensor.
7. The sensor needs to **find the next closest sensor by using the same procedures.** Continue moving and reading all the sensor until the sensorList is empty.
8. **The drone will need to go back to the starting point** if the drone does not move more than 150 times, the drone needs to go back to the starting point. When the drone needs to return to the starting point, the drone can also get an ArrayList<Integer> to prevent the drone's next Position from entering the no-fly zone. Therefore, a similar procedure is used to calculate which direction and the next Position is best to move the drone towards it until it reaches the starting point (within 0.0002 degrees).

- Strategy: How to avoid the no-fly-zone and boundary

Since we have the List<List<Point>> variable and the current position and the next position (where the drone will move). Check if any lines intersect by using the Line2D.linesIntersect method. Treat the line between the current position and the next position as a single line and divide List<list<Point>> into multiple lines. Check if any lines intersect by comparing each line to determine if the next position is allowed.

- Strategy: How to avoid the drone keep moving between two Points and finally spending all the moves or get stuck.

Because we have an ArrayList<Integer>, it stores all accessible directions and its next location. This avoids the possibility of the drone entering the no-fly zone. So, there may be a situation where the direction of the drone is closest to its next location and it doesn't touch any border or no-fly zone. But the next step in this position is likely to get stuck because it is immovable, which usually happens when the drone moves very close to the no-fly zone. The chances of getting stuck would be greatly increased. By analyzing this problem, I decided to avoid this situation by removing the already-moving position. If there is a Position on the path that is the same as the next position of the drone, then this integer is removed from ArrayList<Integer>, so that no moved position exists in ArrayList<Integer>. Instead, the second closest position or the third closest position is selected as the target position. This ensures that the drone does not get stuck.

3.3 Advantage and Disadvantage of greedy algorithms

Advantage:
1. Fast and less computational. I looked up other algorithms such as the two-choice algorithm, the A-star algorithm, and the breadth-first and depth-first algorithms. The time required is one of the lowest.
2. The results of greedy algorithm are good. The average number of steps moved by the drone is around 105, which is good compared to the maximum number of steps moved, which is 150..

Disadvantage:
1. The greedy algorithms will easily get a local minimum result but not the global minimum result.
2. it is easy to get stuck because the drone always looks for and moves to the nearest sensor and does not plan the whole path before moving.

Improvement:
Combining the two-choice and greedy algorithms. I have checked the efficiency of the combination and the average number of moves is reduced to about 90. But sometimes I get confused with this algorithm. Finally, I gave up doing it.

3.4 Images

This image is readings-01-03-2020.gjson.

I forgot about the accurate date of the diagram for the next one. Sorry I didn't save the previous version of my code. But I want to point out the drone get stuck and keep moving between two points.

3. References

https://docs.mapbox.com/
https://geojson.org/
https://tools.ietf.org/html/rfc7946
https://en.wikipedia.org/wiki/GeoJSON
https://en.wikipedia.org/wiki/2-opt
https://en.wikipedia.org/wiki/Minimum_spanning_tree#:~:text=A%20minimum%20spanning%20tree%20(MST,minimum%20possible%20total%20edge%20weight.&text=There%20are%20quite%20a%20few%20use%20cases%20for%20minimum%20spanning%20trees.