Informatics Large Practical

2019-2020

PowerGrab Report

Rares Tudorache

s1729140

CONTENT

1. Software Architecture Description

I chose to do the implementation of my drone by using a collection of six Java classes. Therefore, the project contains the main class, **App.java**, which is responsible for essential tasks like parsing the map and getting all the features together, using data structures like arrays and Array Lists. At the beginning of this class, global declaration of the arrays can be found, which are very useful for taking all the features of the geojson maps. Moreover, in the ***App.java*** class, there is the Main method, where I parse all the arguments and specify the drone mode and where I also chose to generate the required TXT and GEOJSON files. Furthermore, there is the ***Direction.java*** class which is straightforward. This is where I chose to define all of the 16 directions by making use of the special Java type, Enum, used to define collections of constants. Very related to that, there is the ***Position.java*** class which is essential for declaring and initialising(using constructor) the features of one position (latitude and longitude). Moreover, this class handles methods for calculating the next position and for checking if the position is in play area.

Going further, I built the ***Drone.java*** class which is a superclass where I declared and initialized as global variables all the drone features. Both methods and variables are declared as protected as I would like them to be accessible only by the subclasses. This class also contains all the common methods for the implementation of the drone modes, both stateless and stateful. Inheriting that, there are the ***Stateless.java*** and ***Stateful.java*** classes. The Stateless class is one of the subclasses of the Drone superclass and it contains methods specific to the stateless “strategy”, such as moving and handling directions. Then, there is the Stateful class, the most important class, where I implemented the stateful drone strategy using a lot of helping methods for things like choosing the target or avoiding “dangerous stations”. I will go into more detail about that in the next sections.

As a conclusion, this is the system architecture that I went for. I believe it is a really simple, but efficient one. Even if it doesn’t contain such a large number of different classes, it is well structured with logical relations between classes. The Drone superclass and its subclasses Stateless, Stateful and Position were enough for implementing all the required methods and features for solving the task. I am aware that more classes like Station.java or Grab.java could be added for optimising and reducing redundancy, but my architecture with fewer, but more important classes, fitted the task very well and therefore this is why I identify this architecture as being the right one for my application.

2. Class documentation

I will start my description with the main class:

I. App.java

This class begins with the declaration and initialisation of the arrays responsabile for *latitudes*, *longitudes*, *power* and *coins,* which are the main characteristics of the drone that I will be constantly using among the whole implementation. Their predefined length is 50 since this is the total number of features available on a geojson map. There is the ArrayList *path* as well, which is an ArrayList of Points responsible for creating the required geojson file, including the path of the drone after the game is over. Also, there is a random variable of type Random which will be initialized with the given seed. All the methods from this class throw **IOExceptions** in order to handle possible exceptions. The first method of the App.java class is:

- **public static String parseMap(String mapString) throws IOException {**

- this is the main method for taking the geojson map from the web server to the local application. The String parameter that it takes is a link to the map. It sets a HTTP URL Connection with the server and by using the “GET” request method, it returns another String that would basically be the content of that URL. It is structured as a FeatureCollection which in our case, it is a collection of 50 Features. For dealing with this, we needed another method and this is where the getFeatures method becomes useful:

- **public static void getFeatures(String mapSource) throws IOException {**

- this method takes care of the FeatureCollection structure that each geojson map has. It populates all the arrays that I have declared globally using specific methods such as “getProperty()” or “coordinates()”. It takes as a parameter the String that is returned by the previous method (parseMap) and it doesn’t return anything since its purpose is to populate the arrays and get them ready for implementation of the task. Therefore, I created a FeatureCollection using “fromJson(mapSource)” that helped creating a List of Features. Then I managed to create an object of type Point which was needed for taking the coordinates as longitudes and latitudes by initializing the specific arrays using “coordinates().get()”. Furthermore, I initialized the coins[ ] and power[ ] arrays taking

their properties by the “getProperty(“ ”).getAsFloat()” method. Following this, there is the **main** method as well:

- **public static void main(String[ ] args) throws IOException {**

- this is the main method of the application which allows it to run given a specific input. I declared all the required variables and then I parsed the arguments as it was written in the specification document. Here, I also initialized the initial position with the given coordinates. Then I made use of the two methods that I have just described above, parsing the map URL to the specific arrays.

- in the second part of the method there is an **if statement** responsabile for running the desired drone mode, followed by specific instructions that will generate the required TXT and GEOJSON files. For the TXT file I created a PrintWriter that will get as printing argument a variable “outputTXT” which I modify each time my drone has to move. More details about this will be found in the Drone class. For generating the GEOJSON file, I had to “recreate” the entire feature collection and to also add the path of the drone. I used the same reasoning as in parsing the map. I have created a list of Features and I have added a new Feature which is a LineString that contains the generated path. Then, same as for the TXT file, I have taken a variable “outputGeojson” which I have initialized with the resulting Feature Collection and gave it as argument to the Geojson writer.

II. Direction.java

The Direction class is a very simple but essential class. It is of type **Enum** which helped me define the collection of the 16 Directions. Later in the implementation, using “Direction.values()” I managed to get all the Directions as required.

III. Position.java

This is another relatively short but very important class. It extends the superclass Drone and it is mainly responsible for deciding the following position of the drone. At the beginning of the class, there are position’s coordinates and also a List of Directions which I initialize with all the directions from the Direction.java class using “Arrays.asList(Direction.values())”. This is very convenient since it allows us to work with directions much easier.

- **public Position(double latitude, double longitude) {**

- this is the constructor of the class which will initialize the given position with the desired coordinates.

- **public Position nextPosition(Direction direction) {**

- this method is responsible for calculating the next position, given a direction. Using the List of Directions which is declared at the beginning of the class, I get the index of the given direction using “indexOf(direction)”. One thing to mention here would be that the order of the directions in the Direction class matters since I take the index of each direction and I use that further. Therefore we get an **int** that can be used, together with the multiplying variable, to calculate the corresponding degree. Then, I use the “toRadians(degree)” method that converts the degrees into radians. By taking “Math.sin” and Math.cos“ respectively multiplied by the drone **radius**, I manage to calculate the new coordinates. Finally, I create a new object of type Position, initialize it with the new coordinates and return it from the function.

- **public boolean inPlayArea() {**

- as the name suggests, this method checks if the coordinates are “valid”. Given the game boundaries, this method ensures that the longitudes and latitudes are not out of area. It is of type boolean, so it will return true if the coordinates are inside of the area and false otherwise.

IV. Drone.java

The Drone class is the superclass of the architecture. It holds information about all the drone characteristics like *radius, dronePower, grabDistance* and it also initialises the number of moves, *nrMoves,* and *droneCoins.* This class contains all the methods that are common for the implementation of the two modes of the drone. Therefore, methods like connecting to a station, getting the closest station or moving randomly will be found in this class. Moreover, being a superclass, I chose to declare all variables and methods as **protected** so that they can only be accessed by the subclasses.

- **protected static void positiveCollect(int k) {**

- this method takes as input an integer which will represent a positive station that the drone needs to connect to. It will take all the required actions that needs to be done when the drone will get in its range. Therefore it is of type **void** since it will just update the required fields and not returning anything. As a result, when the drone needs to connect to a positive station, the methods will increase its power and its coins accordingly and it will set the station's characteristics to 0.

- **protected static boolean stationInRange(Position nextPos, int s) {**

- this is a straightforward boolean method. It takes as inputs one position and one station and returns true if the station is in range of that position and false otherwise. This is helpful for further implementation.

- **protected static int getClosestStation(Position nextPos) {**

- as the name suggests, this method returns the closest station from a given position. It is one of the most important functions in my implementation. Firstly, I declare an ArrayList<Double> for storing all the distances and also a HashMap<Double, Integer> that would help storing the value(Integer) of the smallest key(Double). I iterate over all stations and for each station that is in the range of the current position, I calculate its distance, add it to the ArrayList and also put it in the HashMap with the corresponding station. Therefore, after all iterations we get an ArrayList populated with all distances within the range. We sort this and then, using the HashMap, the method returns the value of the smallest distance, which will be the index of the closest station. In case there is no station in the position's range, I will return -1.

- **protected static boolean noNegatives(Position pos) {**

- this method helps the drone avoiding the negative stations. It is a boolean method that returns false if there is a negative station in the range of the current position. This method will return true if in the range of the position there is either a positive station or no station at all.

- **protected static int getRandomWithExclusion(Random rnd, int start, int end, ArrayList<Integer> wrongStations) {**

- this is another helpful method for avoiding negative stations. It will return a random integer from the given range excluding the ones that are present in the ArrayList.

- **protected static ArrayList<Integer> badStations() {**

- this is the method responsible for creating the ArrayList that the previous method will take as an argument. I iterate over all possible directions and check for all possible future positions. If there is one position that will be out of play area or that will get the drone next to a negative station then I will add it to the ArrayList and return it. Next, there is the *moveRandom()* method that puts them all together:

- **protected static void moveRandom() {**

- it starts by making use of the two methods above and then it takes the position for the "random" station. Essentially, if the drone moves random it will not land outside the play area or right next to a negative station. Therefore the drone will only move towards "valid" positions.

V. Stateless.java

This is one of the classes that extends *Drone.java.* In this class I have implemented the way the stateless drone should work. It only contains methods that are required for the stateless implementation and it doesn't have any global variables declared.

- **protected static int nrFeaturesinRange(Position nextPos) {**

- I have started with this simple method that takes the current position and returns the number of stations within its range. This will be useful for handling all the cases where the drone might end up in the range of two stations.

- **protected static void moveStateless(Direction d) {**

- this method handles the movement of the stateless drone, given a direction. It updates the drone *power* and *coins* accordingly and it also updates the *outputTXT* variable which is used for generating the TXT file. This is very similar to the *moveStateful(direction)* method, but I chose to do two different methods, rather than just one, for further optimisation and it was also more convenient for debugging.

- **protected static void startGameStateless() {**

- this is the method that starts the game for the stateless drone. It loops until the drone runs out of power or moves and it iterates over all directions. It takes the next “virtual” position, checks if it is inside the playing area and pass that position and direction to the *goStatelessGo(position, direction)* method which will handle the drone’s behaviour.

- **protected static void goStatelessGo(Position nextPos, Direction d) {**

- this method is responsible for the behaviour of the stateless drone. It gets passed from the previous method a position and its corresponding direction and it takes the number of features and also the closest station within the range by calling the *nrFeaturesInRange(position)* and *getClosestStation(position)* methods. Then, it splits the task into three cases by the number of features:

1. If there is only one station in the range of the current position that is positive then it just goes to it and grab its power and coins. Otherwise, it moves “random” using the *moveRandom()* method described above. Also, everytime the drone moves, it updates the *path* by adding the current move.
2. If there is no feature within the range, just move random and update the path.
3. If there is more than one feature within the range, check for the closest one. If it is positive, move there and update the required fields. If the closest station is negative then once again, go to a random valid position and update the path.

VI. Stateful.java

As expected, this class is responsible for the stateless drone behaviour. It extends *Drone.java* as well and it contains only the methods that concerns the stateless drone. Same as the *Stateless.java* class, it doesn’t have any global variables.

- **protected static Direction getNextDirection(int station) {**

- a very important method for the implementation of the stateful drone is this one. It takes a station as input and it iterates over all possible future positions. Once it gets the minimum distance from the station to the positions, it will return the corresponding direction.

- **protected static int targetStation(Position pos) {**

- this is another essential method for the task. It is built in a very similar manner as the method above. It has a position as a parameter and it returns the closest positive station

from the given position. This is called the target, and it will be used for deciding the next direction that the drone should take at each step.

- **protected static boolean oppositeDirections(Direction d1, Direction d2) {**

- this method is straightforward. It takes two directions and checks whether they are opposite or not. I define opposite as being 180 degrees away from each other. I chose to implement this method to get me out of a loop that the drone was getting stuck into. I will talk in more detail about this in the *Stateful drone strategy* section.

- **protected static void startGameStateful() {**

- last, but not least there is the method responsabile for running the stateful drone. It starts by calculating the target. As in the other case, it loops until the drone runs out of power or coins. As long as there is a target (there are positive unvisited stations on the map), it gets the direction of the target and it moves towards it. Then it checks if the the drone reached the target. If that is the case, then it connects to it and it calculates the new target, otherwise it loops again and it keeps going. Once there are no more positive stations to go to (target == -1) then it moves random avoiding negatives.
- I also use an ArrayList of Directions to keep track of the directions that the drone is taking. If the drone keeps moving between two opposite directions, it will move “random” to get back on the desired track.

3. Stateful drone strategy

I chose to implement a Greedy algorithm that at each step it takes the best direction to go. I believe this is a simple and effective approach that could be improved later in the process.

In order to get the best direction, the drone needs to take the direction of a positive unvisited station as a target. Therefore, I had to implement some helping methods for doing that. As a result, the *targetStation(position)* method is necessary. Given a position, look for the closest positive station. With the stateless drone we didn’t have this possibility since it is *memoryless* and it is limited in looking-ahead. In order to search for the closest positive station, the stateful drone has to check all of the 50 stations and calculate the distance for each positive station. Once it gets the shortest distance, it returns the station that corresponds to that distance. This is called the target. If there is no target, the method will return -1. Next step was to get the direction of this

target so that the drone would know where to go. This is where the *getNextDirection(station)* becomes useful. It takes the target as input and checks for every future position if it is valid (if it is inside the play area and if it is not in the range of a negative station). Once this is done, it computes the euclidean distance between all the valid positions and the station and it gets the shortest one, returning its direction. Therefore, the drone will now know where it should go, managing to avoid negative stations as well because of the *noNegatives(position)* method that I used in the method for getting the correct direction.

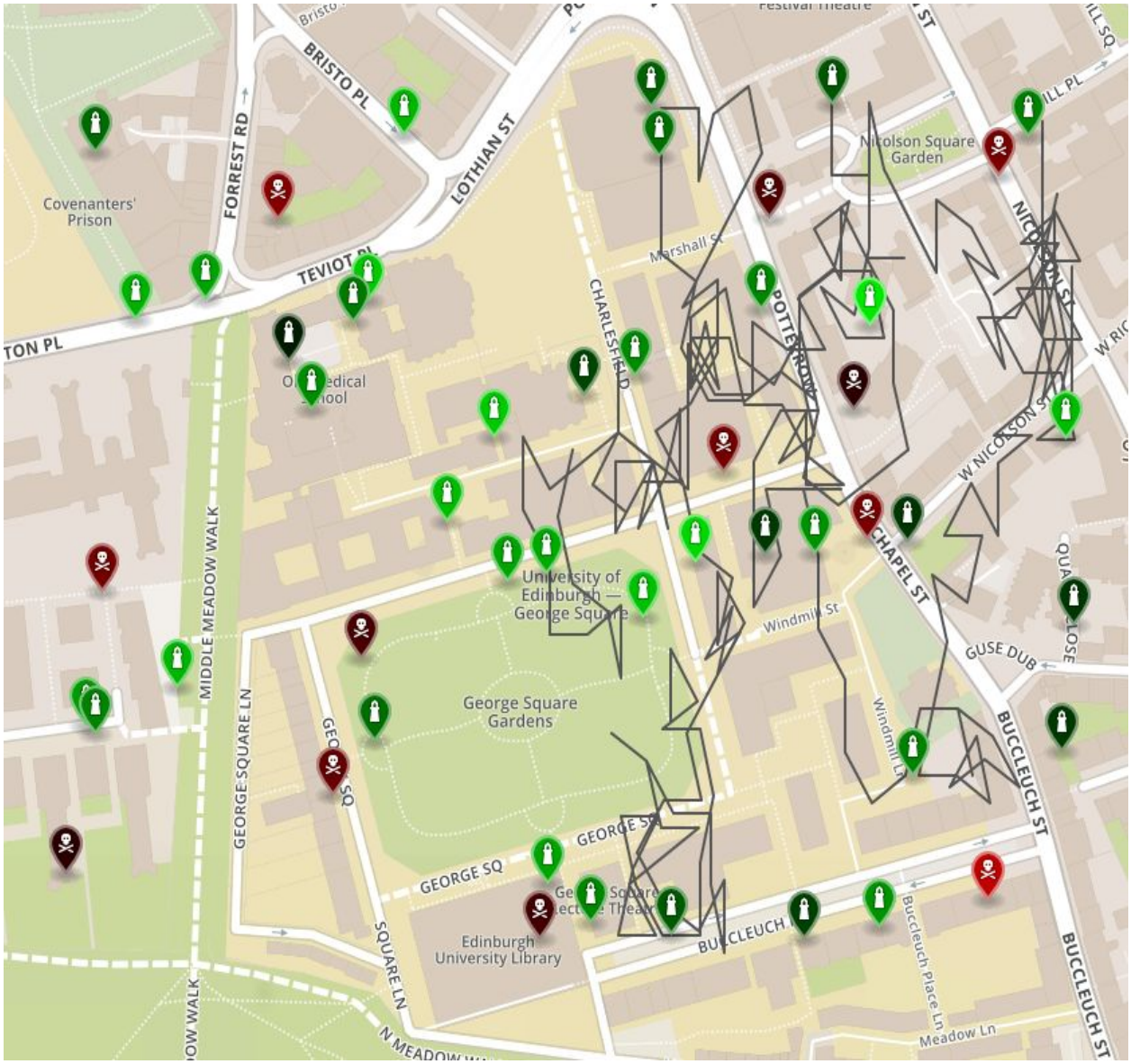
The *startGameStateful()* method is responsible for starting the game. Firstly, it computes the initial target, given the initial position. Then, it starts by getting the direction of target using the methods just described above. The *moveStateful(direction)* method will be used to make the drone move in that direction without landing in the range of a negative station or outside the play area. Once the drone reached its new position, it uses the *getClosestStation(position)* method for taking the closest station within the range of that position. This method was used in the implementation of the stateless drone as well since it only looks at the stations within the range, which is allowed for the both types of drone. At each step we need to check if the drone has reached the target. Therefore, we take the closest station from the range and check if it equals the target. If that is the case, then grab its coin and power, reset the station and recalculate the new target (this is the Greedy decision that the drone makes). The algorithm it will keep doing that until all of the positive stations will be reseted (their coins and power will be equal to 0). When there is no other target for the drone, we want it to wander and trying to avoid negatives. So once the target becomes -1, the drone will go to a random valid position. In this way we are making sure that the drone is not losing any coins while trying to finish the game.

While implementing this algorithm I have encountered some issues. On some maps, depending on the position of the stations, the drone would keep moving between the same two opposite positions. This was happening due to the fact that, each two steps, the calculated target would be the same and therefore the directions would be the same.

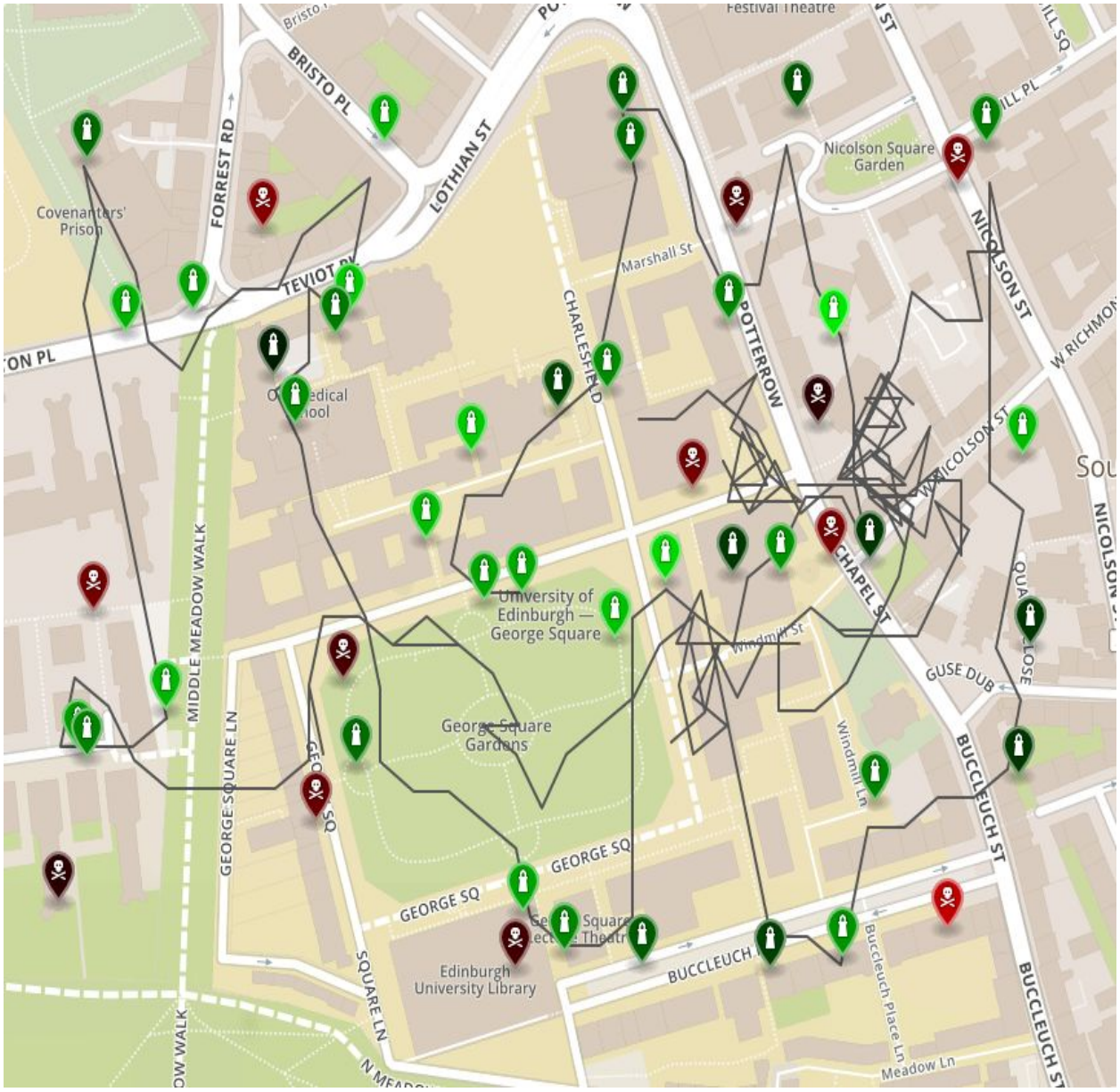
In order to get around that issue I implemented the helping function *oppositeDirections(direction d1, direction d2),* which would check if two directions given as inputs are opposite or not. In addition to that, at the beginning of the *startGameStateful()* method I declared an ArrayList of Directions which will keep track of the directions. Each time the direction of the target is calculated I would add it to the list. Then, I will iterate over the whole list and if there are two consecutive opposite directions, I would remove the direction from the list and I would calculate a new direction for a valid random position. Therefore the drone will get out of that loop and it will recalculate the correct target. But, at the same time we wouldn’t want the drone to

move “random” too often, so therefore I had to find a good trade-off that would work for most of the maps. As a result, I would take a counter and if that counter gets to 100 then the algorithm would never iterate over that list again. I am very aware that this is not the most effective way to fix that issue and that there are some much better ways like the A-Star algorithm, but this is the solution that I managed to implement in the end.

As a conclusion, the stateful drone will always look for the closest positive station and it will try to reach it without flying in the negative ranges. After it has visited all of the targets, it will try to “randomly” fly so that it won’t lose any coins. Comparing to the stateless drone, this is much more efficient and I will use two graphical figures to illustrate that.



Stateless drone 15-12-2019



Stateful drone 15-12-2019

4.Acknowledgements

While working on this project, I was helped by the course lectures and labs and also by the demonstrators. Other than that, I have used different online sources that are listed below.

1. GeoJSON Java Library
   https://gis.stackexchange.com/questions/130913/geojson-java-library

2. “More than you ever wanted to know about GeoJSON”
   https://macwright.org/2015/03/23/geojson-second-bite.html

3. “Generate a random number within a range with exclusion”
   https://forum.processing.org/one/topic/random-int-with-exclude.html

4. “Powergrab visualiser”
   http://geojson.io/#map=17/55.94452/-3.18763