Informatics Large Practical

Implementation Report

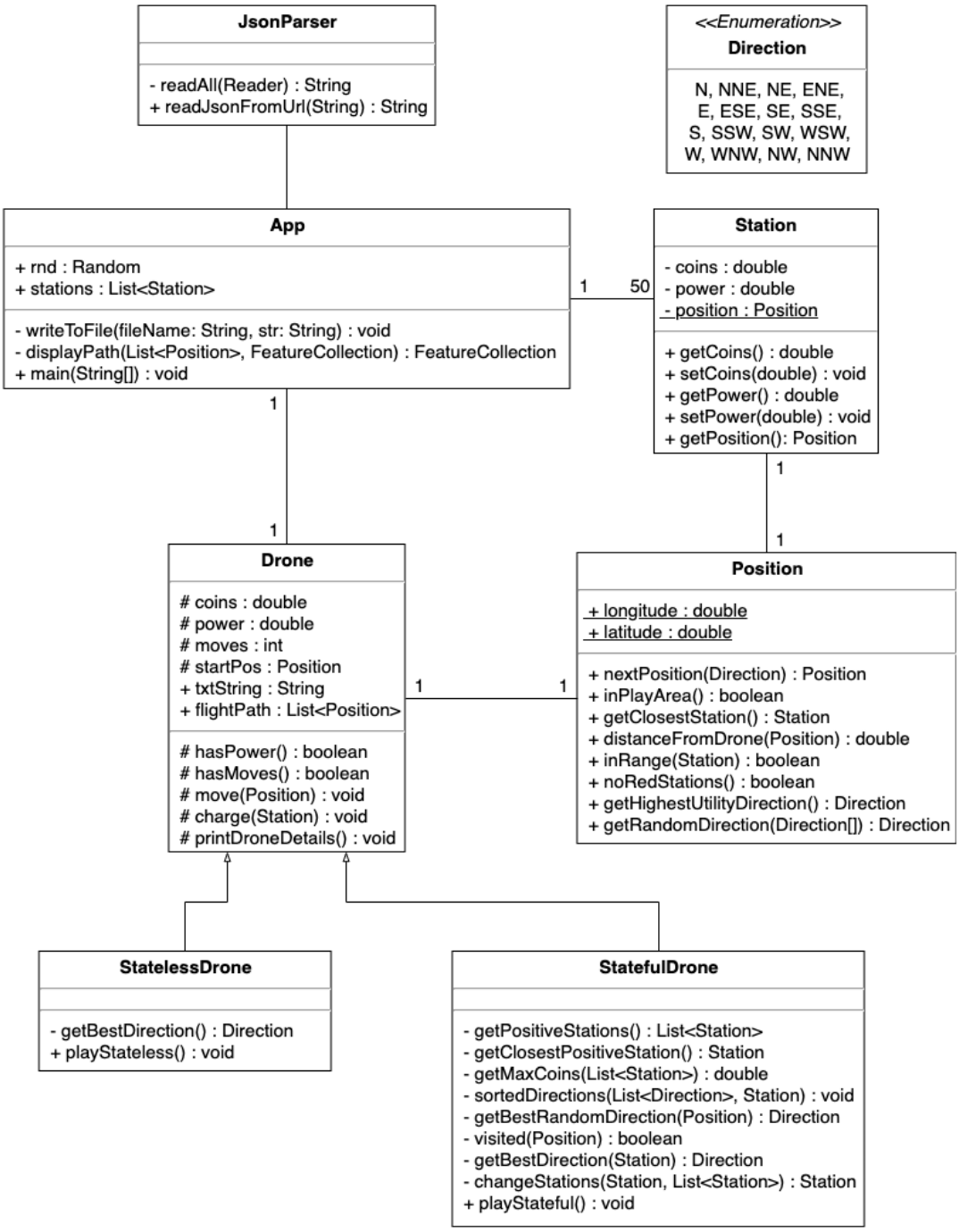
s1709906

Contents

1. Software Architecture Description

In this section I will provide a description of the overall software architecture of my application. I will explain why I identified the Java classes I've chosen as being the right ones for my implementation of the Stateless and Stateful drones. The UML Diagram in the following subsection will show the hierarchical relationships between the classes and provide a brief glance of the methods and variables present in each class as well as their respective visibilities.

1.1. UML Diagram



1.2. Reasoning

The **App** class serves as the main entry point of the whole application. This is where the input arguments are parsed and where the output files are generated. The **App** class also checks whether or not the input arguments are valid or not. Since rendering a powergrab map involves handling JSON data from an external URL, The **JsonParser** class handles the Java networking operations and passes the result as a String object to the **App** class. The **App** class has the Random object variable set by the input seed as well as a list of stations (`List<Station>`) available within the map. These variables are public as they are accessed from multiple external classes (like the **Position** class, for example).

The **Direction** class contains all the possible directions as enums and does nothing else.

The **Station** class represents a **Station** object within the map, and it has the respective coins, power, and position attributes. The attributes of the **Station** object are private, therefore there are getter and setter methods. Since the position of a station should not change, it is a private final variable which does not have a setter method.

The **Position** class holds all the methods that involves the **Position** object, such as determining if the drone is within the play area, checking the closest station from the drone's current position, and so on. The methods in this class are used across the **Drone** classes. The **Position** object itself has the variables longitude and latitude, which are both private final double variables, as they are not supposed to change throughout the course of running the Application.

The **Drone** superclass is extended by the **StatelessDrone** and **StatefulDrone** classes which holds the bulk of the drone movement and coin/power collection, and holds the drone's current flight path. The **Drone** object has the protected variables coins, power, moves, and startPos which are modified within the class or in either the **StatelessDrone** or the **StatefulDrone** classes. The **Drone** object has the additional variables **flightPath** which is a list of positions (`List<Position>` the drone has visited and a **txtString** String object which logs the drone's movements and resulting coins and power. These variables are public because they are used in the **App** class to generate the required output files.

1.3. Class Relationships

The classes **StatelessDrone** and **StatefulDrone** inherit from the **Drone** class. This is because the drones have similar functionality even though their strategies differ, i.e. they both have the same allowed number of moves, starting coins and power, power consumed per move, and they both generate a flight path in the form of a list of positions, and charge from the nearest station that is within its charging range.

Running the **App** class will instantiate either the **StatelessDrone** or the **StatefulDrone** object and gets a list of all the 50 stations within the map. Both the **Drone** object and **Station** object has the **Position** attribute. The only difference is that the **Station** object's position is immutable while the **Drone** object's position is being constantly updated after each move.

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. This class has the following variables which are of type `public static` as they are accessed from multiple classes:

- The variable **`rnd`** is a Random object that depends on the input seed argument.
- The variable **`stations`** is a list of stations (`List<Station>`) that holds all the stations within the map.

This class also has the following methods:

- The **`writeToFile(String`** ***`fileName`*****`, String`** ***`str`*****`)`** method takes in two String parameters, and then it opens/creates the file corresponding to the ***`fileName`*** parameter and writes the ***`str`*** parameter to the file. This method generates the appropriate .txt or .geojson file and writes the output of the Stateless and Stateful drones and saves it in the current directory.
- The **`displayPath(List<Position>`** ***`path`*****`, FeatureCollection`** ***`fc`*****`)`** method takes in the drone's flight ***`path`*** (as a list of position objects) and a FeatureCollection object ***`fc`*** where the flight path will be added to. The output of this method is another FeatureCollection object that will be used to construct the output .geojson file.
- The **`main(String[] args)`** method of this class reads the input arguments, and then it will check whether or not the input arguments are valid, i.e. the right amount of input arguments, correctly formatted month and day inputs, correct state, and the input position is within the play area. If the input is invalid, the application will print the appropriate message to the console and terminate. If the input is valid, method will generate the random seed (and assigns it to **`rnd`**) as well as the map GeoJSON URL based on the arguments. It will then retrieve a list of all the stations within the map and stores it in the **`station`** variable. This method will invoke either the method **`playStateless()`** or **`playStateful()`** depending on the input state argument, instantiating the respective **`Drone`** object (**`StatelessDrone`** or **`StatefulDrone`**). Finally, this method will produce the required output files by invoking **`writeToFile()`** and **`displayPath()`** on the flight path and movement log of the drone after it has finished moving.

2.2. Direction.java

This class holds the 16 different directions the drone can move to - `N, NNE, NE, ENE, E, ESE, SE, SSE, S, SSW, SW, WSW, W, WNW, NW, NNW` - as enums. This class is accessed within methods throughout the application mostly by doing `Direction.values()[i]` where `i` is the index of the direction, e.g. `Direction.values()[1]` will return `N`.

2.3. Position.java

`Position` object contains the following attributes:

- The **`latitude`** attribute which is of type double that represents the latitude of an object.
- 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:

- The **`nextPosition(Direction direction)`** method takes in a `Direction` object as an argument and moves the drone from the current position to the position it goes to when it moves to the specified ***`direction`***. This is done by calculating the change in longitude and change in latitude and adding it to the current longitude and latitude values. These new values will then be used to create a new `Position` object and the next position of the drone is returned.
- The **`inPlayArea()`** method returns a boolean to check whether or not a certain position is within the pre-set boundaries, i.e. between (55.942617, -3.192473) and (55.946233, -3.184319).
- The **`distanceFromDrone(Position newPos)`** method takes in a `Position` object ***`newPos`*** and returns the Euclidean distance between the drone's current position and that object (in this case this is usually the position of a station) as a `double`. The Euclidean distance formula to calculate the distance between point x and y is the following:

$$d(x,y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

- The **`getClosestStation()`** method returns the closest station to the drone.
- The **`inRange(Station station)`** method takes in a `Station` object ***`station`*** and returns a `boolean` to check whether or not the station is within range of the drone, i.e. the distance between the drone and the station is less than or equal to 0.00025 units.
- The **`noRedStations()`** method returns a `boolean` to check whether moving the drone to a certain position makes the drone charge from a negative station, causing the drone to lose coins and power.
- The **`getHighestUtilityDirection()`** method gets the direction with highest utility, i.e. the direction that yields the maximum number of coins. This method is used when a drone has nowhere else to go (i.e. surrounded by negative stations and game boundaries) and would rather charge from any random direction rather than getting stuck.
- The **`getRandomDirection(Direction directions)`** method takes in an array of `Direction` objects ***`directions`*** and gets a random direction using the `Random` object based on the input seed argument.

2.4. JsonParser.java

This class handles reading the JSON data from a specified URL (Java networking).

- The **`readAll(Reader rd)`** method returns the object read by a Reader object ***`rd`*** as a `StringBuilder` object.
- The **`readJsonFromUrl(String url)`** method takes in a ***`url`*** `String`, reads the JSON data from that URL, and returns the JSON data as a `String` object to allow manipulation.

2.5. Station.java

`Station` object contains the following attributes:

- The **`coins`** attribute is a double that represents the amount of coins available in the station.
- The **`power`** attribute is a double represents the amount of power available in the station.
- The **`position`** attribute represents the location in latitude and longitude of the station on the map.

All three attributes are `private`, therefore they all have the respective getter and setter methods, i.e. **`getCoins()`**, **`getPower()`**, and **`getPosition()`**. The station's `position` is a `private final` attribute as it should not be able to be changed (the station does not move around), thus only the setter methods **`setCoins(Double coins)`** and **`setPower(Double power)`** exist.

2.6. Drone.java

`Drone` object contains the following attributes:

- The **`coins`** attribute is a `double` that represents the amount of coins the drone has collected.
- The **`power`** attribute is a `double` that represents the amount of power the drone has at a point.
- The **`moves`** attribute is an `integer` that represents the amount of moves the drone has performed.
- The **`startPos`** attribute is a `Position` object that represents the drone's position in the beginning of a move (before moving to a new direction).

These variables are modified only throughout the child classes (`StatelessDrone` and `StatefulDrone`) therefore they are `protected`. In addition, the `Drone` class also has the following variables:

- The **`txtString`** variable is a `String` object that serves as the result string to be written to the output .txt file. The string contains the drone's starting position before the move, the direction chosen, the position after the move, and the updated coins and power. Each move will be represented as one line.
- The **`flightPath`** variable is a list of `Position` objects, i.e. a `List<Position>` object which represents the positions that the drone has visited.

These variables are `public` since they are used in the `App` class to generate the output .txt and .geojson files. The `Drone` class also has the following methods:

- The **`hasPower()`** method checks whether or not the drone has sufficient power to make a move, i.e. has a power of greater than or equal to 1.25 (that's how much power you need to make one move).
- The **`hasMoves()`** method checks whether or not the drone has exceeded the amount of moves it can make, which was pre-specified to be 250 for both the stateless and stateful drone.
- The **`move(Position nextPos)`** method takes a `Position` object ***`nextPos`*** as an argument which specifies the next position of the drone. This method assigns the next position as the current position, reduces the drone's power and adds the drone's moves count accordingly.
- The **`charge(Station station)`** method takes a `Station` object ***`station`*** as an argument which specifies the station where the drone would collect coins and power from. It will also update the station's coins and power in the list of stations to 0.0.
- The **`printDroneDetails()`** method prints the drone's current power, coins and number of moves. This is invoked after each move for debugging purposes.

2.6.1. StatelessDrone.java

The `StatelessDrone` object inherits the attributes from the `Drone` object.

- The **`getBestDirection()`** method returns the direction that yields the highest number of coins. This method also calls the **`noRedStations()`** method to make sure that the closest station from the next position is not negative, and also calls the **`inPlayArea()`** method to check if the next position is still within the play area boundaries. If one of the previous checks return false, the direction will move in a clockwise manner until it avoids the red station and repeats the same process in search of the best direction. If there are no stations in range, the method will pick the random direction that yields the highest utility by returning the value obtained by calling **`getHighestUtilityDirection()`**.
- The **`playStateless()`** method invokes the **`getBestDirection()`** method, gets the next position from the returned direction and uses it as an argument to the **`move()`** method. When a station is **`inRange()`** of the drone, it will then **`charge()`** from the closest station (**`getClosestStation()`**) and collect the coins and power available at that station, as well as update the coins and power of that station to 0.0. The drone will continue moving as long as it **`hasPower()`** and **`hasMoves()`** left. After each move, the **`flightPath`** of the drone will be updated, as well as the output **`txtString`**.

2.6.2. StatefulDrone.java

The `StatefulDrone` object inherits the attributes from the `Drone` object.

- The **`getPositiveStations()`** method returns a list of all the positive stations in the map by evaluating the number of coins and power each station has from the list of stations.
- The **`getClosestPositiveStation(Position *curPos*, List<Station> *positiveStations*)`** method returns the closest positive station to the drone's current position ***`curPos`*** from the list of ***`positiveStations`***. This station is crucial to decide in which direction should the drone move to.
- The **`getMaxCoins(List<Station> *stations*)`** method will return the total number of coins from a list of stations ***`stations`***. The returned value is used to assess the performance of the drone by comparing it to the final amount of coins the drone has collected.
- The **`sortedDirections(List<Direction> *directions*, final Station *cPS*)`** method uses a `Comparator` to sort a list of ***`directions`*** in increasing order of distance to the closest positive station ***`cPS`***, i.e. the first item in the list will be the direction that brings the drone closest to the closest positive station.
- The **`getBestRandomDirection(Position *pos*)`** method returns the best random direction to take from the `Position` ***`pos`***. In case the proposed random direction brings the drone outside of the play area or brings the drone to a negative station, then this method will pick the direction that brings the drone the farthest from the 'bad' position.
- The **`visited(Position *newPos*)`** method checks whether the `Position` ***`newPos`*** has been visited more than once by the drone. This method is used to make sure the drone is not stuck in a loop whilst it is in the process of trying to reach a positive station.

- The **changeStations(Station *currentStation*, List<Station> *positiveStations*)** method returns the next positive station (instead of moving to the current positive station) by removing the ***currentStation*** from the list of ***positiveStations*** and selecting a new positive station, then appending the ***currentStation*** at the end of the list to be 'visited later'. This method is called when the drone is stuck in a loop whilst trying to reach the ***currentStation***.
- Like the stateless drone, the stateful drone also has a **getBestDirection(Station *closestPositiveStation*)** method which returns the direction that brings us to the ***closestPositiveStation***. Obstacle handling is also done in this method, which will be explained in the next section. This method will return `null` if the next position obtained after moving the drone to the selected direction has been visited more than once.
- The **playStateful()** method gets a list of positive stations and will move the drone towards the closest positive station. The method invokes the **getBestDirection()** method, gets the next position from the returned direction and uses it as an argument to the **move()** method. If the next position has been visited more than once, this indicates that the drone is getting stuck and the **getBestDirection()** method will return null and the **changeStations()** method will be called. The drone will now move to the station returned by that method (another positive station) instead of the previous station that got the drone in a loop. If there are no more positive stations in the **positiveStations** list, then the drone will just go to the direction returned by the **getBestRandomDirection()** method. When a station is **inRange()** of the drone, it will then **charge()** from the closest station (**getClosestStation()**) and collect the coins and power available at that station and remove the station from the list of positive stations (if it is a positive station). The drone will move as long as it **hasPower()** and **hasMoves()**. If there are still positive stations remaining on the map, the drone will continue moving towards one of them. However, if the drone still has sufficient power and moves but all the positive stations have been visited, the drone will loop back and forth to the most recently visited position. This loop is chosen instead of moving to a random direction because this will return a much cleaner flight path in the visualizer, which makes debugging and understanding the drone's movements much easier. After each move, the **flightPath** of the drone will be updated, as well as the output **txtString**.

3. Stateful Drone Strategy

The stateful drone implements a much more complex strategy than the stateless drone as it has more capabilities. The stateful drone has the objective function which is to move from one positive station to another to collect the coins and power available at that station. The optimal solution would be one that allows the drone to collect from all the positive stations and avoid charging from all the negative stations. The stateful drone will also avoid obstacles in a much less random way than the stateless drone. This section will discuss how I took advantage of those capabilities to make the drone's coin and power collection much more efficient.

3.1. Getting positive stations

One advantage that the stateful drone has over the stateless drone is that the stateful drone is able to remember the stations that it has visited and can store the information of the stations in the map. Invoking the `getPositiveStations()` return a list of all the positive stations within the map, I would be able to instruct the drone which station it should go to next and keep moving towards the target station until it has been reached, instead of wandering around an empty area and taking random directions until a positive station is detected.

3.2. Moving towards the nearest positive station

After obtaining a list of positive stations within the map, the drone would pick the station that is closest to its current position and assign it as a target station by invoking `getClosestPositiveStation()`. This will increase efficiency by making sure that the drone does not use its power to travel from one side of the map to another to collect from a station, whilst there are other closer positive stations that it has neglected on the way. This station will then be used as an argument in the `getBestDirection()` method to determine which direction will lead the drone the closest to the target station. After reaching the nearest positive station, that station will be removed from the list of positive stations and the drone will find another nearest positive station. By targeting the closest positive station in each move, the drone's flight path will be much cleaner (less haphazard like the stateless drone) and it will use its power much more efficiently.

3.3. Handling obstacles

The `getBestDirection()` method will move the drone to the direction which will lead the drone the closest to the target station. However, most of the time there are obstacles in the way of getting to the target station in the form of play area boundaries and negative stations. If there are no good directions to go to, i.e. the drone is surrounded by the play area boundary and negative stations, the `getHighestUtilityDirection()` method will be invoked.

3.3.1. Play area boundaries

If the next proposed position of the drone leads the drone closer to the positive station but leads the drone outside of the play area boundaries (e.g. because it's trying to find the shortest path or it is trying to avoid a negative station), then the drone would pick the second best direction, i.e. the direction that brings the drone second closest to the target positive station.

3.3.2. Negative stations

There are cases where the targeted positive station is surrounded by negative stations but the drone still wishes to go towards that direction anyway. In this case, the drone will check if the closest station after making the move is a negative station or not. If yes, then the drone would pick the second best direction, i.e. the direction that brings the drone second closest to the target positive station. In one of the first implementations of my strategy, the drone checks if the next position brings it within a charging range of a negative station. However, I realised that being within a charging range of a negative station does not matter as long as the closest station is either a neutral station (0 coins) or a positive station. This method will prevent the drone from moving away from a positive station that overlaps ranges with a negative station.

3.3.3. Loops

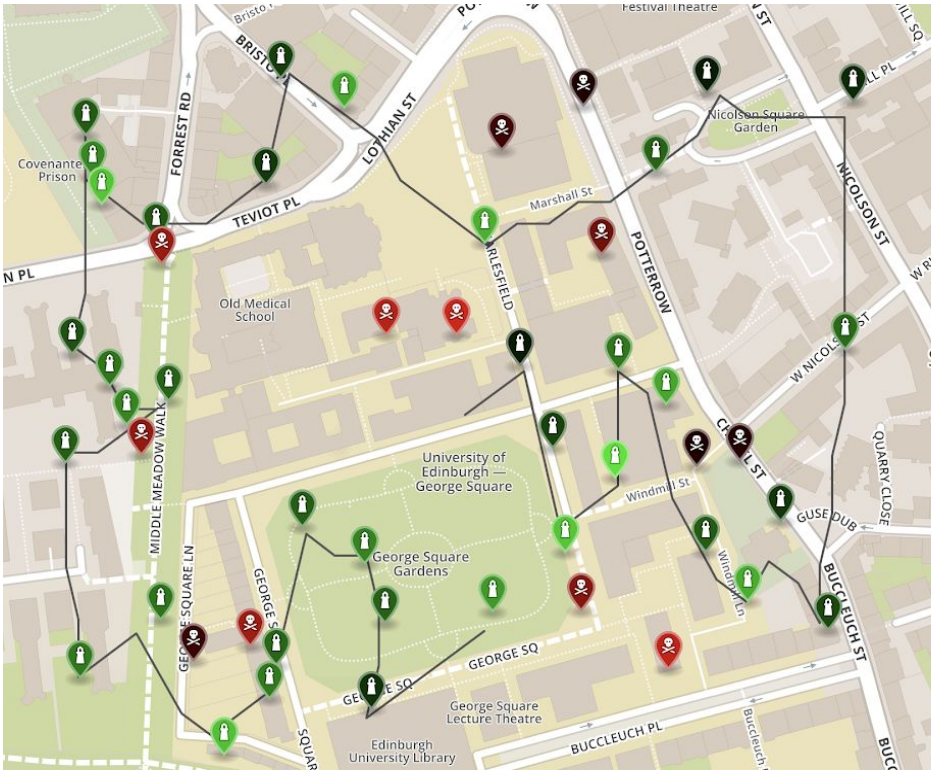
In the rare case that the targeted positive station is surrounded by a play area boundary and negative stations, the drone will loop back and forth - first it would move towards the positive station and then it finds itself in a less than ideal situation (e.g. hitting the play area boundary) and moves away from that. After that it will move towards the same direction it moved to previously as the drone thinks that it brings it closer to the positive station, while in reality it is hitting a boundary.

To prevent the drone from getting 'confused', I added a boolean `visited()` which will return true if the position has been visited more than once and the `getBestDirection()` method will return null. I allowed the drone to visit the same position once in case it is stuck in a position where the only ideal move is to go back to its previous position. If this boolean detects a loop, the drone will then change its target positive station (`changeStations()`) and move to another positive station on the map that has not been visited yet and adds the previous target positive station at the end of the list of positive stations, indicating that the drone will come back to it later. This solution is sub-optimal but I found that this is still much more efficient than moving to any random direction.

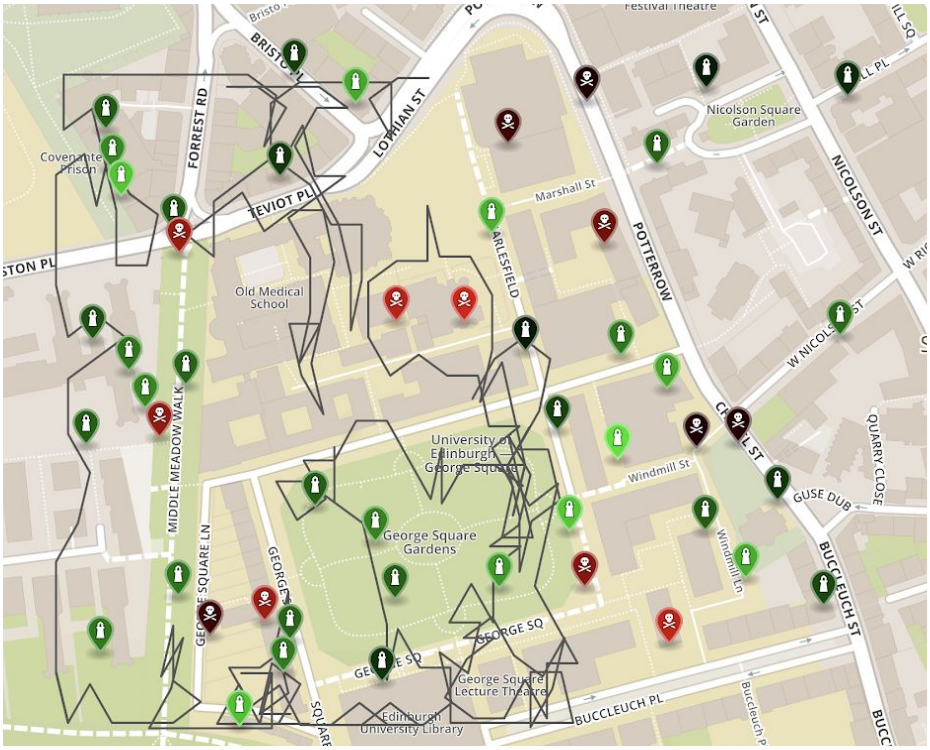
However, one edge case that I found is that this could be a problem if the loop occurs when the drone is trying to charge from the *last* positive station on the map. In such case, the drone will then just move towards any direction that does not lead it to a negative station but is still within the play area boundaries, i.e. the direction returned by `getBestRandomDirection()`.

3.4. Improvement from the stateless drone

The following figures show the flight path of the two drones on the same map:



Stateful Drone on the 10th of October, 2019.

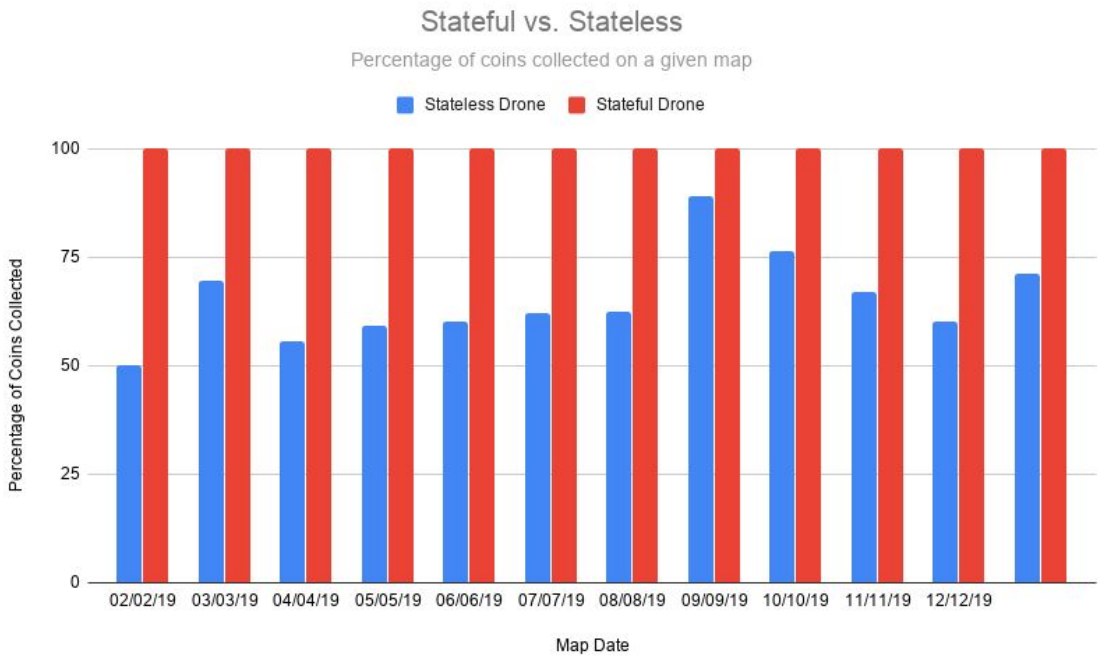


Stateless Drone on the 10th of October, 2019.

From the comparison above it is easy to tell that the stateful drone moves in a much less haphazard manner and has a much more directed movement. This is because the stateless drone has no objective other than moving randomly and avoiding red stations, hoping that the next random move would lead it to a positive station. The stateless drone is not capable of moving towards a target station until it is reached or remembering the direction it previously moved to.

The major difference is that for every move, the stateless drone will pick the best random direction that it can go to, i.e. the one that yields the most coins. It is very likely that the drone will end up being in the middle of nowhere (no stations in range) and decides to move randomly. the stateful drone, however, will keep moving towards the direction that brings it closer to a target station, which - in most cases - means moving in the same direction that it moved to previously if there are no obstacles that would prevent it from doing so.

As a result, the stateful drone is able to collect all the coins from the positive stations within the map, while the stateless drone will run out of power trying to find a positive station without knowing where the station actually is. The following graph compares the percentage of coins collected by the stateless drone to the percentage of coins collected by the stateful drone across 12 different maps:



Stateful Drone and Stateless Drone coin collection percentage comparison

3.5. Testing

To test my drone, I used the **`evaluator.py`** to check the percentage of coins my stateful drone collects from a particular map across over 700 different maps (maps from 01/01/2019 to 31/12/2020). At first, I would check the maps where it has less than optimal performance, i.e. the ones where it obtains less than 95% and I would run the drones on those maps to inspect what really happened and what caused the poor performance.

Through analysing the flight path of my drone on those maps visually, I was able to deduce problems such as the drone getting stuck within a loop, getting stuck in between negative stations, and much more. I would rethink my strategy and think about more edge cases and then I would run my new implementation on **`evaluator.py`**. I kept on improving my strategy until I obtained results that I think are optimal, i.e. 100% ratio with a run time of less than 1 second per map. The run times of each map varies with the machine that I am using to run the script, so I would only double check if there was a case where it would take more than 5 seconds to run my application.

While testing my stateful drone using **`evaluator.py`** I noticed that the code gets stuck due to, for example, not meeting an exit condition in a while loop. Therefore, I duplicated the python script and modified every occurence of ‘stateful’ to ‘stateless’ and changed the name of the output .xlsx file so it doesn’t overwrite the one for stateful. I ran through the values a couple of times and checked if anything looked anomalous and made sure that the code ran on all the maps.

4. References

- Mert Bora Alper's evaluator http://homepages.inf.ed.ac.uk/stg/ilp/
- The JsonParser https://community.apigee.com/questions/52082/how-to-retrive-the-json-data-using-java.html
- MapBox SDK https://docs.mapbox.com/android/maps/overview/
- Java SDK https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html