Powergrab Implementation Report

Contents

0.1 Abstract

The task we were delegated to perform was to implement an autonomous (Non-player controlled) drone that competes against a player in a game called Powergrab. Powergrab is a game played within the confines of the University of Edinburgh central area, and consists in going around and collecting coins from stations randomly distributed across the play area. The collection stations may be negative, and therefore, the object of the game is to go to as many positive stations while avoiding negative stations, in order to achieve the highest amount of coins in the smallest amount of moves.

We are tasked with developing an "easy mode" stateless drone which a novice player could compete with, as well as a "hard mode" stateful drone which is meant to be as efficient as possible in collecting coins in few moves, and is therefore more adequate for a more skill-full player to compete against

The task is to be implemented in Java, utilizing tools from the Mapbox SDK, and be built using the Maven build automation tool.

1 Software Architecture Description

1.1 UML Class Model

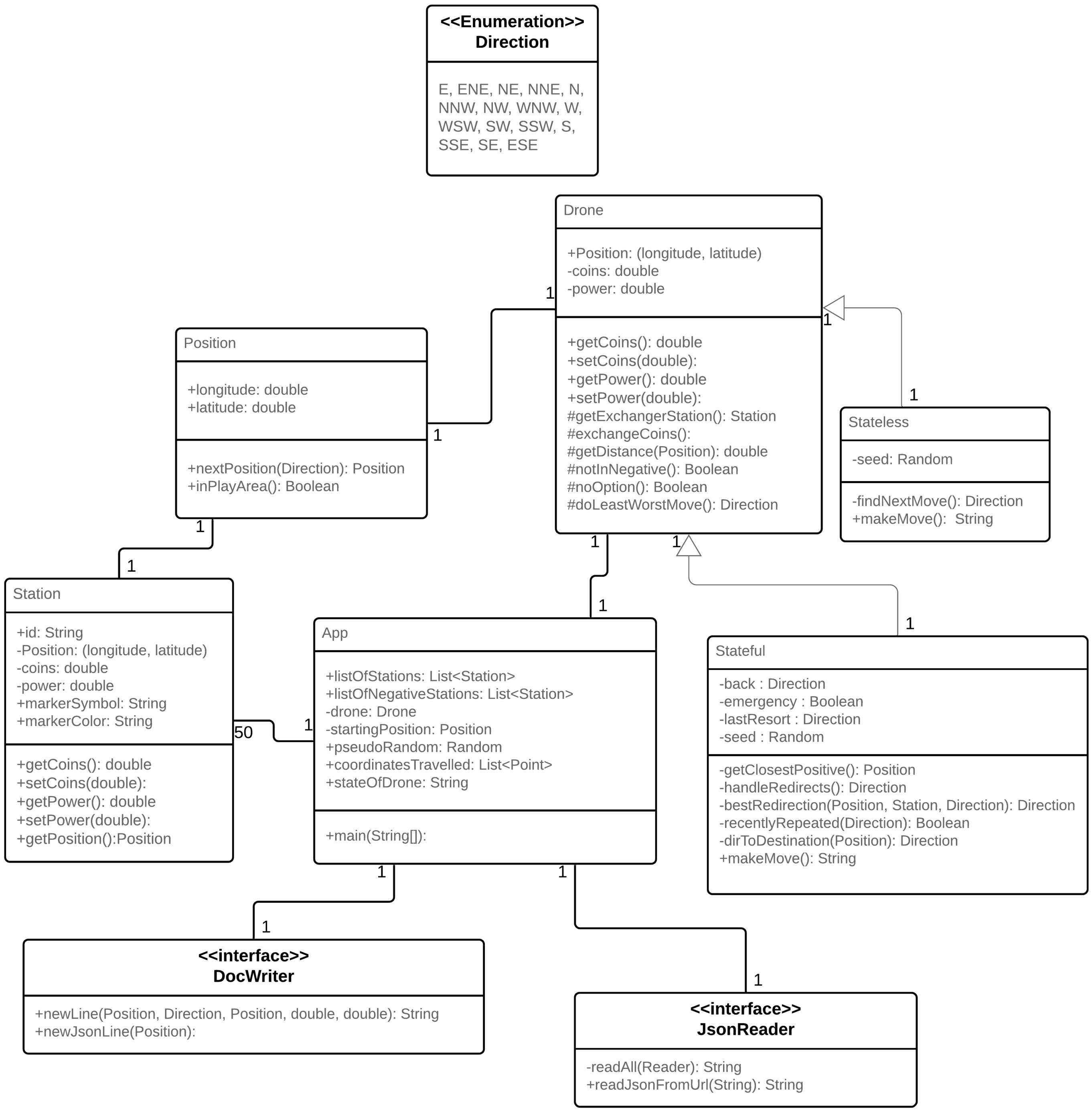

Figure 1: UML Class Diagram Powergrab

1.2 Rationale

App:

The App class is meant to act as a controller for the overall project. It holds the main method which is meant to bring all the different classes, objects and methods together and interact with each other for the sequence flow of the program. Furthermore, it holds a large part of the interaction with the Mapbox SDK and networking necessary for the function of the Geojson aspects of the project. It also initializes important global variables, which are used elsewhere in the program.

Direction:

The direction class is the most simple component of the program, and merely holds the values of all the 16 compass directions that the drone can use for navigation and positional calculations and predictions.

Even though the Direction class is a very small enumeration class, it is still distinct enough from other objects, and has an important role to play in the entirety of the project. Furthermore, it is an enumeration class in which their order affects calculations in the rest of the project. This means that despite being small, having it as a separate class is still of vital importance.

Position:

The position class functions as a bedrock for the interaction between the drone, the stations, and the play area map. Using the Longitude and Latitude which act as 'x' and 'y' coordinates to place the play objects on the map. It allows the drone to stay within the confines of the game while circumnavigating it according to the rules and logic of both the drone types that have been implemented.

Objects of type position can be found in almost all classes since it is a necessity when it comes to placing the game on two dimensions which can be understood and manipulated by all these classes. It is also an important part of the Drone class and Station class, since the means by which these communicate is in terms of proximity and angular directions, which would become impossible without understanding their position relative to each other.

Station:

The stations in the game of Powergrab function as the objectives and the obstacles of the game. They also possess some unique identifiers due to how the Json system works, which naturally necessitates a Station class to translate these Json features into the project, and extract the fields of interest, namely position, power, and coins.

Drone(Stateless and Stateful):

The Drone class possesses two sub-classes, namely Stateless and Stateful. The reason the hierarchy functions in this manner, is that despite the Stateless and Stateful drone being very distinct in terms of the information they have behind their decision making, they also have common fields and methods that they necessarily must share. The Drone class brings together both drone implementations, and allows them to have a common pool of fields and methods while still having their own different logic within their separate sub-classes.

JsonReader and DocWriter:

The JsonReader and DocWriter function as interfaces, and their principle role is to parse through the Json file, and write Geojson/Text files respectively. The justification for existing as separate examples to improve the readability and organization of the project by placing the methods that interact with external files in separate classes.

1.3 Relationships

One of the most notable occurrences in the UML class diagram is that the Direction class is not bounded by any relationships. The reason for this is not to say that the Direction class is isolated from the rest of the program, but is standard for an enumeration class. The reason for this is that enumeration classes function more as data types then concrete classes.

Since the App class is the controller class, it naturally is the most connected class in the architecture of the project. It has a 1 to 50 relationship with the Station class, since their will always be 50 charging stations in the Json file that is parsed. The App class also has 1 to 1 relationships with both interfaces, which is a result of the interfaces not producing objects but rather possessing special methods used to parse Json files, and write text/Geojson files. The App class also has a 1 to 1 relationship with the Drone class, which is necessary since only one instance of the drone can be active during a game session. Even though mock drones are created to perform calculations, structurally speaking only one instance of the drone is active in the life cycle of the program.

The Stateless and Stateful classes have indicators in their relationship arrows to indicate they are child classes of the Drone super-class. This means the public or protected variables and methods are inherited from the Drone class. Which is the structural case of the program in practice. The Drone class also has a 1 to 1 relationship with the Position class. This is due to a drone only being able to have 1 position at a time, even though in the life-cycle of the program it may take up to 250 different positions.

The Position and Station class have a 1 to 1 relationship since each instance of a Station may only have 1 Position.

1.4 Protection

The private fields indicated with a '-' are due to the variable only being used locally in the class. It is important to note that even though Stateless and Stateful are sub-classes of Drone they still don't have access to private variables.

The public fields indicated with a '+' are global variables and methods that can be used throughout the project, and are a necessary part of the interaction between classes, as well as the controller's capacity to bring everything together.

The protected fields indicated with a '#' are used in the case of the Drone class, since this allows sub-classes to inherit methods and variables from the Drone, which should not be accessible to other classes that do not have the Drone class as a parent class.

The Direction class doesn't have protection identifiers since it is representative of a data type rather than variables or methods.

2 Class Documentation

2.1 Direction.java

The Direction class has no methods or variables as it is simply meant to represent a datatype for the 16 compass directions the drones can navigate. The directions are placed in the order representative of a unit circle, starting with east, and increasing counterclockwise in the order E -> ENE -> NE -> NNE -> N -> NNW -> NW -> WNW -> W -> WSW -> SW -> SSW -> S -> SSE -> SE -> ESE -> repeat. This is often used in the project in increments of 22.5 for angle calculations as 16*22.5 leads to a full 360 degrees for the full range of angles in a circle.

2.2 App.java

Fields

- public static List<Station> listOfStations: This is a public variable used to store the 50 stations in the Feature List of the Json file. It is populated in the main method and utilized in the Drone class and its sub classes for several calculations in the life cycle of the program.
- public static List<Station> listOfNegativeStations: As the name would entail this variable stores the stations that have negative coin/power values at the beginning of the program. This variable is populated in the main method and used in the Drone class and Stateful child class.
- private static Drone drone: This is the drone that is worked on in the life cycle of the program.
- private static Position startingPosition: The initial position of the drone.
- public static Random pseudoRandom: The pseudo random number generator used in the Stateless and Stateful drones to implement repeatable randomness for repeat testing.
- public static List<Point>coordinatesTravelled: A list of points which will hold all the points the drone has occupied. It is later converted to a LineString to be added to the feature list and be outputted to the Geojson file.
- public static String stateOfDrone: Stateful or Stateless drone

Methods

- public static void main(String[]): The main method begins by checking if the input arguments are the right length, and terminates the program if not. Then the date, starting position, random number generator seed, and state of drone are retrieved from the inputs. It then checks if the starting position and state are valid, and terminates the program if not. Then the URL where the Json file is retrieved is parsed, and a feature collection is created from it. The list of features, which are all stations are parsed through and their position, coins, id, marker symbol, and marker color are retrieved and made into Station objects. A text file and Geojson file are opened and writers are created for both. Thereafter a 250 move counter is initialized, and the moves are made checking the drone has enough power to make a move. Finally the LineString(drone black box) is added added to the original file, and the text and Geojson file are closed terminating the program.

2.3 Position.java

Fields

- public double longitude: Holds the x value of the position.
- public double latitude: Holds the y value of the position.

Methods

- public Position nextPosition(Direction): This method uses simple trigonometry to return the position of the drone after making 1 move in one of the 16 compass directions.
- public boolean inPlayArea(): This method ensures that a position is within the designated play area of the Powergrab game. This means between -3.184319 and -3.192473 for longitude, as well as between 55.942617 and 55.946233 for latitude.

2.4 Station.java

Fields

- public String id: Unique identifier that each station has. This is very useful for debugging.
- private final Position position: The position that the station is located at. Multiple stations cannot occupy the same position.
- private double coins: This is representative of the potential coin difference the drone will have after exchanging with the station. Furthermore a station can have a maximum of 125.0 coins.
- private double power: This is representative of the potential power difference the drone will have after exchanging with the station. Furthermore a station can have a maximum of 125.0 power.
- public String markerSymbol: Marker symbol of the station. This is only useful for the Geojson file as the drone's logic doesn't deal with the marker symbol.
- public String markerColor: Marker color of the station. This is only useful for the Geojson file as the drone's logic doesn't deal with the marker symbol.

Methods

- public double getCoins(), public void setCoins(), public double getPower(), public void setPower() and public Position getPosition(): These are the getters and setters for the stations. Get means retrieve, and set means modify. The position variable only has a getter since the position of the station cannot change over the course of a game.

2.5 Drone.java

Fields

- public Position position: This is the position the drone occupies.
- private double coins: This functions as the score of the drone, and can be a minimum of 0.0. Even if a station has more debt to give the drone, if the drone is at 0 the exchange stops. Initialized to 0.
- private double power: This functions as the fuel of the drone, and can be a minimum of 0.0. If the drone attempts to make a move with less than 1.25 power, it will fail and the drone will stay still till the game is over. The Geojson and text file should reflect this. Even if a station has more debt to give the drone, if the drone is at 0 the exchange stops. Initialized to 250.

Methods

- public double getCoins(), public void setCoins, public double getPower() and public void setPower(): These are the getters and setters for the drone. The position can change and future positions can be predicted so a getter or setter for these field would be poor practice.
- protected Station getExchangerStation(): This method retrieves the closest station within 0.00025 degrees distance, and is used to predict outcomes if the drone were to move in a certain direction, and also to actually perform an exchange after a move. It does this by parsing through all the stations and updating the selected station variable when it has a smaller distance than 0.00025 and a smaller distance then any other previous stations. Both child classes(Stateless and Stateful) use these methods.
- protected void exchangeCoins(): This method calls the getExchangerStation method and uses the retrieved station in order to perform an exchange between the exchanger station and the drone. The drone cannot take power or coin values below 0, so if a station is more negative then the drone is positive, the stations coins and power will increase till the drone reaches 0. Otherwise the drone will have the coins and power as a result of the sum of the station and the drone, and the exchanger station will have 0 coins and 0 power. Both child classes(Stateless and Stateful) use these methods.
- protected double getDistance(Position): This method takes the position of a station and returns the distance between that station and the drone. This method is used in other methods within the Drone class as well as in both child classes.
- protected Boolean notInNegative(Direction): This method takes a direction, and creates a drone instance for the resulting position of the drone heading in that direction. Then the negative stations are parsed through, and the getExchangerStation() is called to check if the new drone will exchange with any negative stations, which is obviously undesirable. This method is used in other methods within the Drone class as well as in both child classes.
- protected Boolean noOption(): This method checks if all directions a drone can take result in negative exchanges. This is useful should a drone spawn surrounded by negative stations/edge of the map, as it avoids the logic of the drone being unable to select a move.
- doLeastWorstMove(): This method is called in both child classes when noOption() returns true, and checks which move is legal and results in the least amount of coin loss.

2.6 Stateless.java

Fields

- private Random seed: This is retrieved from the app class and is the pseudo random number generator used for controlled randomness. It is important in this class since the stateless drone makes a lot of decisions based on this randomness.

Methods

- private Direction findNextMove(): This method represents the bulk of the logic behind the stateless drone operation. It first checks noOption() and if it doesn't have options calls doLeastWorstMove() and ends the move. Otherwise it proceeds by creating an array of directions which result in negative exchanges or the drone being out of play area. After this it parses through the stations that it is close too and checks if it can perform a move such that it will be able to exchange with the station. It then takes the direction that results in an exchange with the most positive station and returns it. If it cannot make an exchange with a positive station it will return a random direction that is not within the array of negative and out of play area directions.
- public String makeMove(): This method is called in the main method of the App class for every move of a stateless drone. It finds the direction of the next move, calls the nextPosition() method to update the drone's position, and then exchanges the coins. It also adds the new Point to coordinatesTravelled by calling the newJsonLine() method from the DocWriter class (to be explained further in the document). The String it returns pertains to the string that will be written in the text file. Furthermore

2.7 Stateful.java

Fields

- private static Direction back: This field keeps track of the previous direction, and is used in the emergency mechanism when the drone has cornered itself. It is also more predominantly used to go back and forth once positive stations run out.
- private static Boolean emergency: This is set to true if the drone has made a move where every move besides returning results in a negative exchange.
- private static Direction lastResort: Once the drone has returned from where it cornered itself, this direction will note which direction resulted in a cornering, so the drone can ignore this direction once it makes a move after returning from being cornered.
- private Random seed: This holds a pseudo random number generator for the few instances where the stateful drone uses randomness.

Methods

- private Direction dirToDestination(Position): This method uses basic trigonometry to compute the optimal direction to a given position. It uses the desired position as well as the position of the drone, and uses the angle between them to get a optimal direction to that destination.
- private Position getClosestPositive(): This method parses through the list of stations, and returns the Position of the closest positive station. It also checks if the positive stations have run out and simply returns the opposite of the previous direction if this is the case,

resulting in the drone moving back and forth. Then the method checks if the drone can exchange with its closest station with any direction, and returns that direction if so. Otherwise the drone simply uses the dirToDestination() method to return the direction that most efficiently travels to the closest positive. If the drone is going to repeat, meaning it is stuck going back and forth from already visited stations, it goes in any other direction such that it is is still in play area and isn't exchanging with a negative station. This allows it to approach the station from a different angle to make the exchange.

- private Direction handleRedirects(): This method first checks noOption() and calls doLeasWorstMove() if it has no options. Then it returns the direction of getClosestPositive(). Then it checks if the move from getClosestPositive() results in a negative exchange. If it does it enters bestRedirection() to get a correct direction.
- private Direction bestRedirection(Position, Station, Direction): This method takes the desired direction as a starting point, and is called when the drone heading in that direction will result in a negative or illegal position. It takes the angle to the positive station and the angle to the negative station it is hitting and uses this to compute the most efficient path to go around. the negative. If the angle to the positive is larger than it means it needs to go around in a counter-clockwise fashion, and vice versa. If going in that direction results in a repetition, it means a cluster of negatives is creating a repetition problem, so it will simply go around the longer way to avoid this complication. If the move has no moves except going back, then it will say there is an emergency(set emergency to true) which allows the drone to return and then for the next move go though the logic but without going the direction that resulted in the emergency.
- private Boolean recentlyRepeated(Direction): This method takes the positions of the last 11 positions of the drone and checks if the drone is about to enter that position again. This is method is used in many ways to avoid situations where the drone ends up repeating itself.
- public String makeMove(): This method calls the handleRedirects() method to get the direction of the next move. It performs the move and then exchanges coins. It also takes note of the position before to write the new text file line and Json Point for the eventual LineString feature.

2.8 DocWriter.java

Methods

- public static String newLine(Position, Direction, Position, double, double): This method takes the position before the move, the direction the drone went, the position after the move, and the coins and power after the move and returns a string displaying all of these. This is used to write a line in the text file.
- public static void newJsonLine(Position): This method creates an object of type Point from the coordinates of the drone's position before the move, and then adds it to the List of Points coordinatesTravelled(final Point added after moves loop in main method). The reason for this is that a List of Points can be converted to a LineString which can be made into a feature. This feature can then be added to the original Json file and the final Geojson file will have the stations and the path of the drone.

2.9 JsonReader.java

Methods

- private static String readAll(Reader): This method adds the text of document character by character to a string.
- public static String readJsonFromUrl(String): This method takes a URL and then retrieves the text from this URL and the readAll() method is called to retrieve the entire file as a single string.

3 Stateful Drone Strategy

3.1 General Outline

The stateful drone strategy consists of going towards the closest positive station while avoiding negative stations in a manner that minimizes the number of moves wasted in avoiding them. Once all positive stations have been visited, and the coins in the map have been completely exhausted the drone is meant to pace back and forth, waiting for its power or the its remaining moves to terminate.

The stateful drone strategy seems simple when being summarized concisely, but naturally involves several nuances and edge cases that lead to more complex algorithms and ideas taking part, these are delineated below.

3.2 Finding the Closest Positive Station

The first step in implementing the strategy is to give the stateful drone the capacity to pinpoint the location of the closest station. Because of the relatively small area of play, the use of the great circle distance is an unnecessary complication, and we will therefore treat the play area as a flat surface. This leads to a relatively simple calculation, as it is simply the euclidean distance between the points.

$$d(u, v) = \sqrt{(u_x - v_x)^2 + (u_y + v_y)^2} \tag{1}$$

Where x is the longitude of v, and y is the latitude.

Therefore, we simply parse through the our positive stations and use the positions associated to the smallest euclidean distance.

3.3 Direction to Closest Positive Station

The directions available to us cannot be represented as a continuum, but as 16 discrete directions associated with a standard compass. Therefore we cannot perfectly direct ourselves towards a station, but with a simple manipulation of angles we can find the optimal cardinal direction to the station.

We can imagine a standard unit circle where angle 0 lies on the x axis between quadrant 1 and 4 on the coordinate plane, where the angle increases in the counter-clockwise direction. We can equate this to our compass where 0 is East, 90 is North, 180 is West, 270 is South and all 16 directions represent an increase of 22.5 degrees. Therefore we can get the angle between the drone and the station we are directing ourselves too, and simply get the difference between that angle and the angle represented by the direction. Finally if that difference is less than 11.25, we know that is the optimal direction.

3.4 Going Around Negatives

When directing itself to a positive station, the drone will naturally have to around any negative stations that stand in its way. So finding the quickest way to get around while remaining on course is of great importance for the performance of the stateful drone.

When dealing with this problem there are three important variables to consider; the position of the drone, the position of the negative station, and the position of the positive station desired. Again, we have to compute the angle between all these components to define the optimal route for the drone to take.

We first get the angle between the drone and negative station, and then the angle between the drone and the positive station. We then compare these to decide the trajectory. If the angle to the negative station is less than the angle to the positive station, and their difference is less than 180 degrees then this means the most efficient travel path for the drone will be to go around in a positive manner(counter-clockwise in the unit circle). If the angle to the negative station is larger than the angle to the positive, then the drone goes around in a negative(clockwise) manner.

3.5 Dealing with Multiple Negative Clusters

Often, when pathing around a negative station the drone may run into a wall formed by two or more negative stations. This can lead to complications where the drone may want to go clockwise around the first station, and then counterclockwise for the next, essentially cornering the drone in a pocket. This can also occur if the drone wants to go around a station, but the radius of the negative station leads to outside the play area.

Because of the way the drone travels optimally, we know that if the drone runs into the situation, the drone will simply repeat itself and go back and forth believing it is going in the best direction over and over. So in order to deal this we need to be able to detect if the drone is repeating itself and act accordingly. We can simply save the position of the most recent moves, and detect when a move the drone has selected will result in one of these positions repeating themselves.

When a repetition is detected, the drone will simply do the reverse of what it would normally do, that is to say it will go around the long way to the positive station, instead of pacing back and forth. This means that if the angle to the positive station is greater than the angle to the negative, the drone will go around counter-clockwise, and vice versa for the reverse case. This will always be optimal if the drone is going to be outside the play area, and is mostly optimal when the drone is cornered between two negative stations.

3.6 Avoiding Zeroes

Zero stations, that is to say stations that have already been traded with present a very special kind of problem. They are obviously harmless when directing the drone towards a positive station, but may result in the drone attempting to collect coins from a positive station but ending up closer to the zero station. This means that zero stations should only be taken into account when the drone can arrive at the desired station within the next move.

One can imagine three stations close together at the same latitude.The drone approaches them directly from the West. The drone goes East, collecting coins from the first station, then East again to collect coins from the second station, but ends up closer to the third, then attempts to go west in order to collect from the second but ends up at the first again. This leads to the drone repeating itself till moves or power are exhausted.

For this reason in our stateful algorithm we first check if the ideal direction will result in an exchange with the desired station. If not, we check if there is any other direction the drone can head towards, where it will exchange with its closest positive station. If so, it will proceed in this direction no questions asked. If not, the algorithm moves on to the next sequence of checks and calculations.

3.7 Emergency Mechanism

Because of the way we do not allow for repetition, there is a possibility the drone may go in a direction, and then have nowhere to go without ending up in a negative station. If this is the case, we trigger an emergency mechanism were we allow the drone to go back to its previous direction, and remove the direction that resulted in the emergency from the options for one move. This means the drone must choose to head a different way, and the drone proceeds as normally thereafter.

3.8 No more positive stations

Once all positive stations have been visited and exchanged with, the drone will simply move to the previous position. This means the drone will arrive at the final station, and then begin pacing back and forth. Since the drone was just recently at that position, we can know with confidence that the previous position is not a negative station or outside the play area, and it is therefore safe to just go back and forth till power or moves run out.

3.9 No Options Available

There is a possibility where a drone will spawn completely surrounded by negative stations. In order to deal with this, we need to be able to react to this possibility without entering an infinite loop.

Should the drone be unable to move without going to a negative station or outside the play area, the drone will select the direction that leads to the least negative station that is still within the play area. This is an unlikely scenario but is a necessary consideration to ensure that the code is robust and doesn't fail to run completely.

3.10 Stateless Example

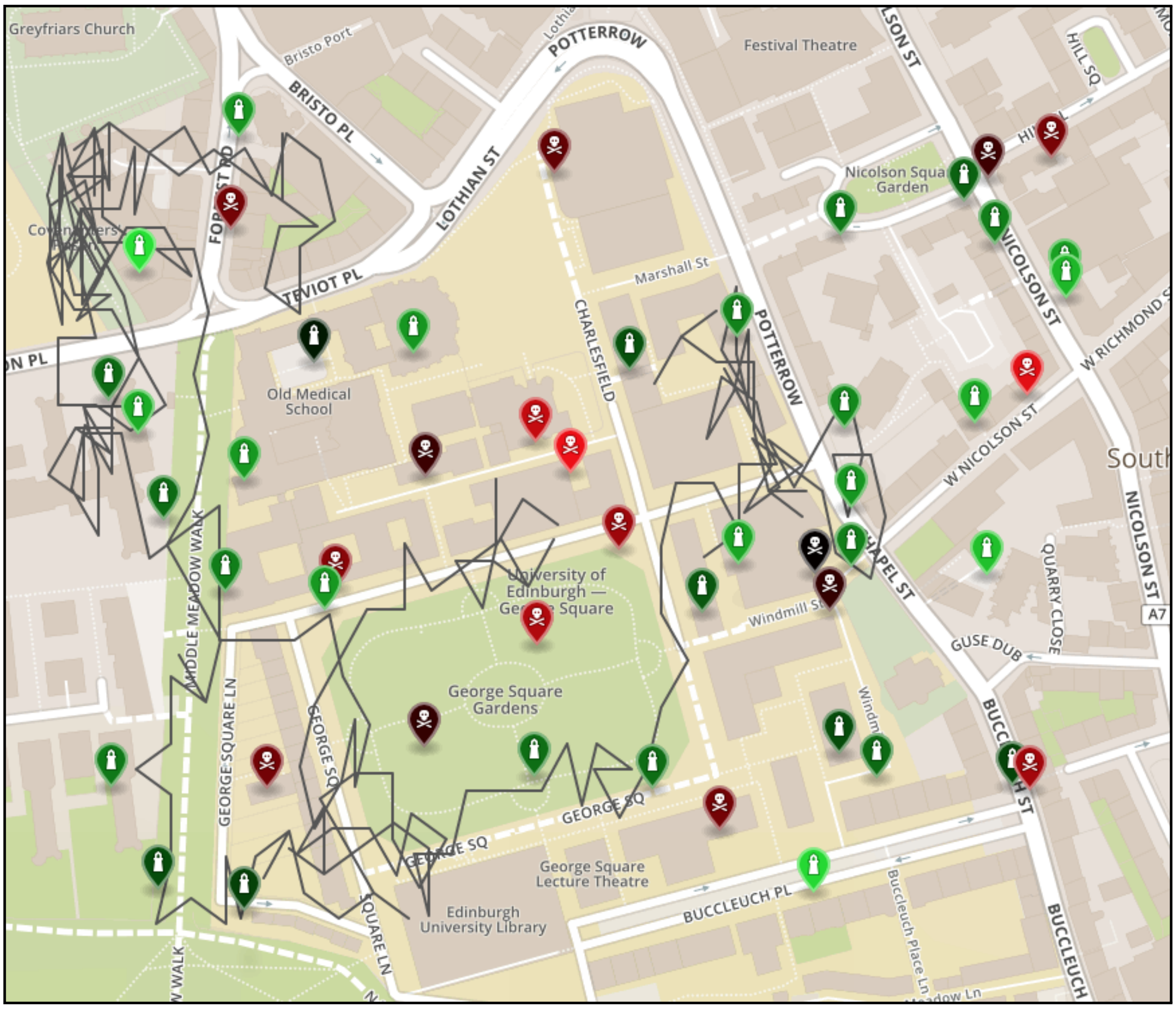

Figure 2: 11 09 2020 55.944425 -3.188396 5678 stateless

3.11 Stateful Example

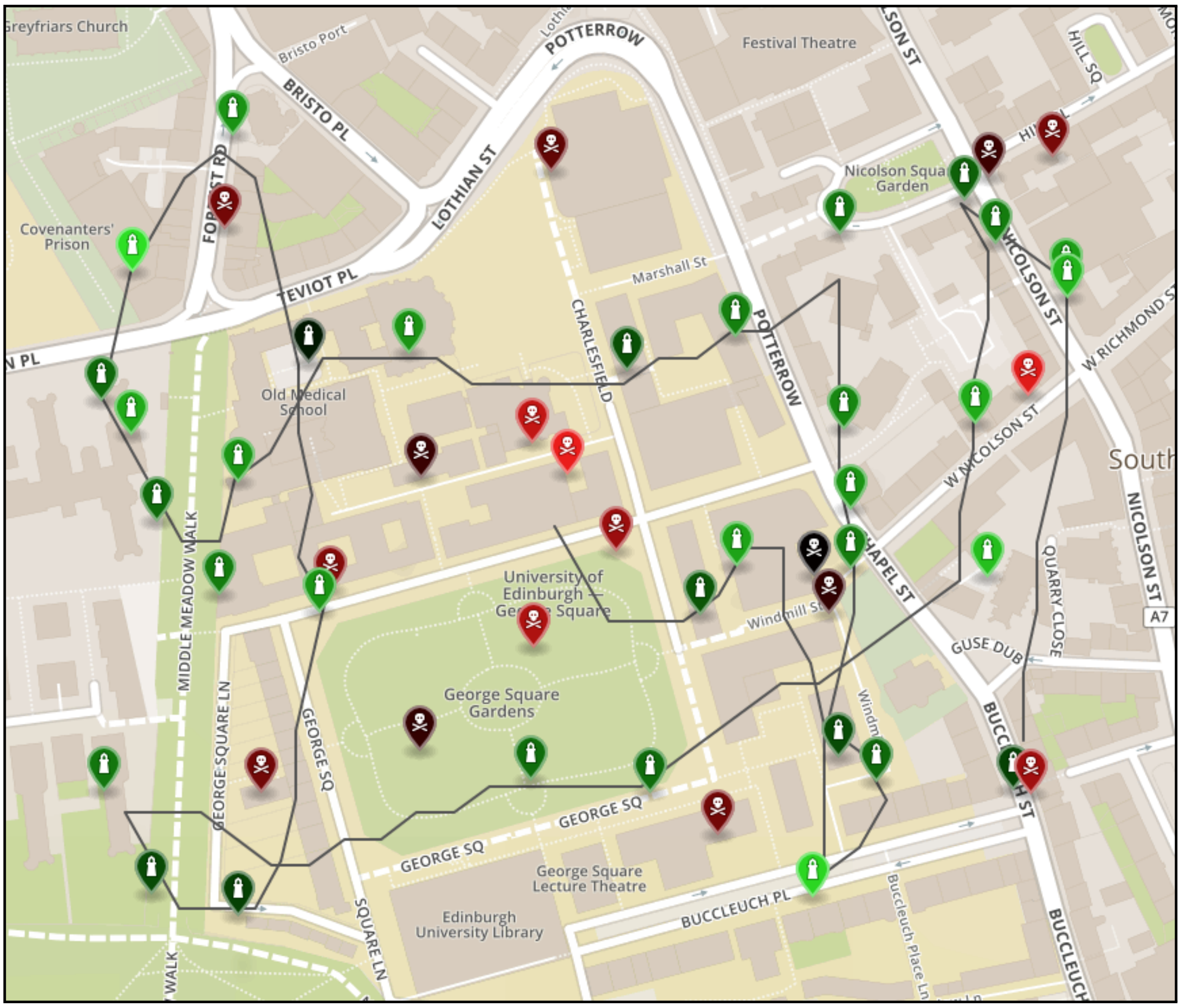

Figure 3: 11 09 2020 55.944425 -3.188396 5678 stateful

References:

- "Glossary." ROSALIND, Rosalind, http://rosalind.info/glossary/euclidean-distance/.
- "GeoNet." Distance on a Sphere: The Haversine Formula | GeoNet, The Esri Community | GIS and Geospatial Professional Community, ESRI, 5 Oct. 2017, https://community.esri.com/groups/coordinate-reference-systems/blog/2017/10/05/haversine-formula.
- Ceta, Noel. "All You Need to Know About UML Diagrams: Types and 5 Examples." Tallyfy, Tallyfy, 25 Aug. 2019, https://tallyfy.com/uml-diagram/.
- Stephen Gilmore and paul Jackson Slides and Coursework Docs