Informatics Large Practical Report

Table of Contents

1 - Software Architecture Description

Overview of software architecture

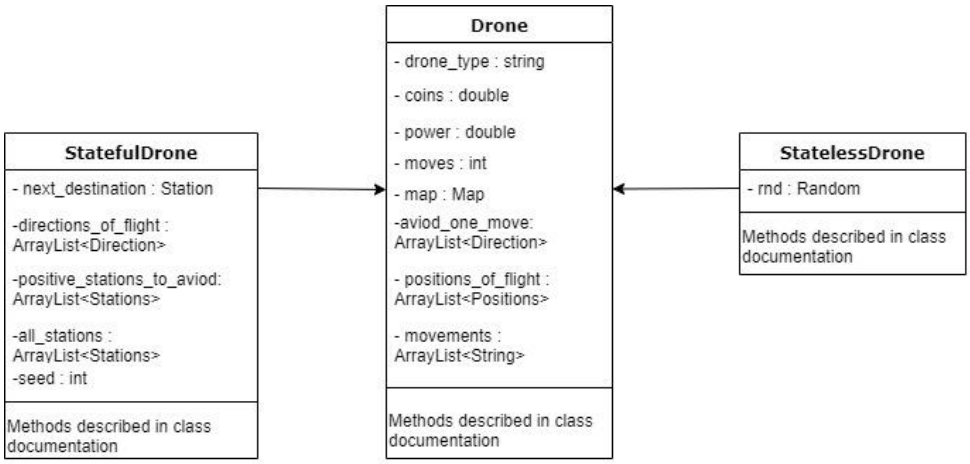
Object-Oriented Programming attempts to model real world problems, separating objects in the real world into different classes to help developers understand the code more intuitively. They will be able to identify and understand how your program works easily.

In the game one can immediately identify some core components. The drone, the stations and the map. These ultimately define the game and are 3 of the classes in my software architecture. The drone can be split into 2 types, stateless and stateful. The drone must always have a position and have a general direction in which to go in. Other classes that where made where vital for running the game itself and producing the output files (JSON and text).

Drone

I will discuss the Drone, StatelessDrone and StatefulDrone classes.

The drone class is used to represent the drone in the game. This is one of the most important classes since this is the entity which plays our game. There are 2 types, stateless and stateful. The fundamental difference between them is the way they move around the map, their strategies. But they share all functions that drones have, that is why I decided to make a class hierarchical relation between the Drone class and StatefulDrone and StatelessDrone classes respectively. They are both subclasses of the super class Drone.



The Drone class, thus the StatefulDrone and StatelessDrone too, must have a map associated with it and a position. This was a natural way to define a drone because each one created is placed in a specific map and always has a specific position within that map. Thus, a drone depends on the Map class and Position class.

The StatefulDrone has a closely linked class, this is the SimulateStatefulDrone. I created this class because the strategy I made for the stateful drone required to “simulate” runs and thus it must create various versions of itself and do different strategies to find the best one. The StatefulDrone class in the strategy uses the SimulateStatefulDrone.

SimulateStatefulDrone

The reason for creating this class was due to the strategy the drone takes. Before the drone flies the drone must find which one of its three strategies maximizes the total amount of coins in the map. Therefore, it was natural to create a class that has the exact same information about the drone and the amount of moves it wants to take in each map. This class allows the drone to test various strategies to find the one that helps in get the best result.

The elements that are in this class are *Day, Month, Year, latitude, longitude, drone_type* which are the same of the drone we are trying to simulate. This means that this class is completely dependent on the drone that it is trying to simulate.

Station

In the game the drone flies around the map picking up coins and power with the objective of the getting as many coins as possible. The place where the drone picks up or loses either coins or power is at a station. As a result, the Station class representing these points on the map was made. The Station gets its attributes from the feature collection of the map however the main attributes of the features we are interested in are the coins, power and position of the station.

As a result of this the Station class has a Position attribute which is always the same, and unique for each station on each map. This class is also vital for interactions with the drone since the drone will charge and pick up coins at these locations.

One would think to consider making a subclass of negative and positive stations. However, I didn't make these classes, this is because the core difference between them is that the negative stations have negative coins and power (they subtract from the drone) and the positive add to the drone's totals. And so, as a result the variables for these 2 are only negative and positive and so we can easily distinguish between them since positive stations have a positive amount of coins and power and negative stations the opposite.

Position

The Position Class is also very important because it defines the position of any entity in the game. The drone must have a place on the map which it belongs to and so do the Stations. This class not only provides the position (latitude and longitude of a drone or station) but is also used to calculate distances and another metrics which can be used by the drone during the game. For example, if it is the play area or if it is in range of a station.

Direction

This class is used to identify the 16 possible directions the drone can travel in. We must define in what directions the drone can travel in because this is how it must get around the map. This is because when we calculate a new position of a drone once it moves, we must know where it is in the map according to what direction it goes in.

Map

The drone is playing in a certain map with unique locations and attributes of stations; the map class is used to get the map that the drone will be playing in. It was vital to have a Map class; this is because the drone must play in a certain area with defined positions for the stations. So, having this class will allow the drone to know where it is playing and where stations are in this map. It will contain the Day, Month, Year of the map and a list of all the features and stations. The list of features is the same as the list of stations but is used for the output file. This class does depend on the Station class.

FileCreator

This class is used for the output files (JSON and text). It was necessary to have this class outside of the drone and other classes, this is because the FileCreator is not in itself the way the drone works. It has nothing to do with the game itself, but rather it is used to produce the output of the drone. It was important for this to be a class to get the results of the drone.

Hence this class depends on a Map and a Drone. This is because to produce the text file we need to know all the movements of the drone and write them in a readable fashion for a user. The drone keeps track of all its movements and so this class will be able to access them and print them out.

App

The App class is the framework where the drone will run. This class is identified because we need to have a place where the simulation framework of the drone can take place. This is where the game will run. It will read 7 command line arguments and then produce the right map and drone from this information. Then it will run the drone on this map and get the results back of the flight.

2 - Class Documentation

This section will be useful for a developer if they want to continue with the project this is because I will provide explanations for each class and the methods within them. Furthermore, this will allow a developer to understand the reasoning for each method and how they work. As well of the use of variables class and instance variables in each case.

Classes

Abstract class Drone

The Drone class is an abstract class which represents a drone in the game. It is the superclass of the drones. The reason for it being abstract is because we will never have a Drone playing a game, we will have a StatefulDrone or StatelessDrone, so this prevents the class from being instantiated. The super class will have all the variables and methods that both of the 2 types of drone have in common. This will prevent repetition of code and allow us to use functionality of them between the classes by taking advantage of inheritance.

Variables

This class has no class variables. This is because every time we have a new drone, we will have to rest the variables within this class and as a result none are unique to a drone, all depend on the drone that is being created.

The instance variables for the drone are:

1. *drone_type* : this variable is a String and keeps the type of the drone, stateless or stateful.
2. *coins, power* : keeps track of the coins and power of the drone both type double.
3. *moves* : keeps track of the number of moves a drone takes. Integer.
4. *map* : the current map that the drone is playing in. Type is Map.
5. *avoid_one_move* : this variable resets after each move of the drone, it keeps track of the directions that the drone should not go in with one look ahead. This includes directions where the drone falls in range of a negative charging station.
6. *rnd:* type Random which is used to produce a number between 0 and 15, it is used for the drone to find a direction when it doesn't know where to go. It is used in both drones.

The following 2 variables are used for the output files of the drone they are vital that we keep track of them because in the drone class since we need to do it for both drones. Note that neither of the drones will use these variables when flying, they are only used for the FileCreator class.

7. *positions_of_flight* : This is an ArrayList of positions (latitude, longitude) of the drone. It keeps all the positions the drone goes to during the flight. This is used to produce the trace over the GeoJson file that represents the path of the drone.
8. *movements* : An ArrayList of Strings this will keep a list of strings, these strings are in the from: "55.944425,-3.188396,SSE,55.944147836140246,-3.1882811949702905,0.0,248.75" this is the format needed for the output text file. So, if the drone does 250 moves it will keep strings of this from in order from first to last which will be easy to print to a text file later.

Constructor - *public Drone(String DD, String MM, String YYYY,double lat, double longi, int seed, String drone_type)*

This class only has one constructor. This is because we need all the information from the command line to produce the correct drone. We need to have the right map and type of drone it will be.

The drone constructor takes the Day (DD), Month (MM), Year (YYYY) of the map and creates the appropriate map by making a Map object in the constructor and assigns it to the *map* variable. Sets the *coins* to 0 , *power* to 250.00 and *moves*

to 0. Sets the *drone_type* to the drone type given in the constructor and creates a Position object using the latitude and longitude given, this will be assigned to *current_position*. The seed is used to make the *rnd* variable.

Methods

The methods used by the drone are all non-static, this is because they must all be used by the drone object and would not make sense if they are used in a static sense.

1. *public abstract void run(int moves) :*It takes an integer (moves); this is the number of moves the drone will run for and returns void. This is because it allows the drone to move around the map with its strategy. This method is where the logic for the run of the drones will be in.
2. *public void one_look_ahead() :* It doesn't take any inputs and returns void. It updates the *avoid_one_move* variable. This method is the one look ahead of the drone. This function must be called after every movement a drone takes this is because it must reset the variable *avoid_one_move.* It checks all the directions the drone can take and sees if the drone in a given direction it falls in range of a negative charging station. If it does it adds it to the *avoid_one_move* variable.
3. *public void update_drone(Direction direction):* It takes as input a Direction (direction) and it returns void. This method moves the drone in the direction given and updates variables: (a) power it subtract 1.25 from it. (b) moves adds one. (c) current_position it changes it to the new position resulting from moving in the direction given.

   And then checks if it is in range of a station to transfer the coins and power to the drone and calls *transfer()* method. Lastly it creates the string of the movement, the string that will be used in the text file in the output and adds it to movements variable.

4. *public Station find_station_transfer(ArrayList<Station> stations):* It takes and ArrayList of Stations and returns a Station. This method finds the closest station which is in range for transfer. This is to ensure that after the drone moves to its new position it checks that if it is in range of a station it will transfer. If no stations found will return null.
5. *public void transfer(Station station) :* This method takes a Station and returns void. The method ensures the transfer with the station and then updates the drone and the station appropriately. It will add to the coins and power of the drone the values of coins and power of the station.
6. *public boolean inPlayArea_and_aviod_negative_station(Direction direction) :* This method takes a Direction and returns a boolean (True or False). This method is used to check if a drone takes a particular direction it will remain in the play area and it doesn't fall in range of a negative station. It uses the *inPlayArea* and checks that the direction given is not in the *avoid_one_move* variable.
7. *Getters and Setters :* These are all the getters and setters for the private variables. We have getters and setters for: drone_type, coins, power, movements and positions_of_flight. We only have getters for map, current_position and rnd. They are defined by the following convention: get or set + name of variable with capital first letter. For example, the getter and setter for coins would be *getCoins()* and *setCoins(double coins).*

public class StatelessDrone extends Drone

This is the stateless drone class; here we have the unique functionality that the drone takes. It inherits all the methods and variables for the abstract class Drone, so it makes use of the functions and variables in the methods in the class.

Variables doesn't have any new ones, all inherited from super class Drone.

Constructor - *public StatelessDrone(String DD, String MM, String YYYY, double lat, double longi, int seed, String drone_type)*

It is the same as the Drone constructor so calls super(DD, MM, YYYY, lat, longi, drone_type) to use the super class constructor but has one new variable which is the seed which is used to make the Random rnd. Throws exception because of the super() call since we are making a map which could have some problems with the connection to the URL.

Methods

The methods that the Stateless drone are boiled down to 3. The *run(int moves)* method which runs the drone for a given number of moves and 2 other methods *direction_one_look_ahead()* and *final_direction(Direction direction)* which are used to find the direction the drone should move in using the stateless drone's strategy.

1. *public void run(int moves) :* This method takes the number of moves you want the drone to do (int) and then makes the drone fly around the map for the given moves using the stateless drone's strategy. It is a simple while loop that ensures that the number of moves taken by the drone is less than the ones set, and the power of the drone is always greater than 1.25 if it fails any of this conditions this means that either the drone has finished or run out of power so it stops.
2. *public direction direction_one_look_ahead() :* This method doesn't take any parameters and returns the direction the drone should go in (can be void). It iterates over the 16 directions and if it finds a direction in which in falls in range of a positive charging station it returns this direction. Goes to the first one it finds.
3. *public direction final_direction(Direction direction) :* This method takes a direction and returns the final direction the drone should go in. If the direction is not null it checks that going in the given direction doesn't fall in the range of a negative station or falls out of the map, if it satisfies this condition goes this direction. However, if the direction is null it will pick a random direction to go in with the *rnd* variable.

public class StatefulDrone extends Drone

This is the stateful drone class which we have the logic of the stateful drones' tactics. It inherits all the methods and variables for the abstract class Drone, so it makes use of the functions and variables in the methods in the class. However, it has quite a few more variables that it keeps track of to get better performances.

Variables

It inherits all the Drones variables has a few extra ones; this is because the stateful drone is much more complex than the stateless drone. They are all instance variables.

1. *next_desitiantion* : type Station, this keeps track of the next station the drone has to go in, will be the same until the drone visits the station.
2. *directions_of_flight* : ArrayList of Directions , this is all the directions the drone goes in this is useful to see if the drone gets stuck (repetition of positions).
3. *positive_stations_to_aviod* : ArrayList of Stations, this is all the positive stations that fall in range of a negative charging station, used in *strategy_2*.
4. *all_stations* : ArrayList of all stations in the map.
5. *seed* : Integer seed given to constructor, used for making the SimulateStatefulDrone object.

Constructor - *public StatefulDrone(String DD, String MM, String YYYY, double lat, double longi, int seed, String drone_type)*

It is the same as the Drone constructor so calls super(DD, MM, YYYY, lat, longi, drone_type) to use the super class constructor but has one new variable which is the seed which assigns this value to *seed*. For *directions_of_flight* and *positive_stations_to_aviod* variables it creates 2 empty ArrayList and for *all_stations* we can get it from the map.

Methods

The methods that the Stateful drone all non-static. They are the logic for the stateful drone's tactic.

1. *public void run(int moves) :* This method takes the number of moves you want the drone to do (int) and then makes the drone go around the move around the map for the given moves using the drone's strategy. It is a simple switch statement which picks the strategy the drone will implement after the drone calls *simulate()* method form the SimulateStatefulDrone class.
2. *public void strategy_*(int moves)* *(1,2 or 3) : This method runs the drone for the given number of moves using the stateful drones * strategy. It is a simple while loop, that encapsulates the strategy taken every move, ensures that the number of moves taken by the drone is less than the ones set, and the power of the drone is always greater than 1.25.
3. *public void update_drone(Direction direction) :* This method overrides the Drones update_drone method, it does the same as the original one but adds the direction taken to the *directions_of_flight* , this is so we can keep track of the directions that the drone goes in.
4. *public boolean is_stuck() :* This method doesn't take any parameters but returns a boolean. Checks if the last 10 movements of the drone are being repeated if that is the case then the drone is stuck so returns True otherwise returns False.
5. *public void stations_to_aviod() :* This function finds all the positive stations that are in range of a negative charging station and then adds them to *positive_stations_to_aviod* this is only used in *strategy_2()* before the while loop.
6. *public void find_station() :* This function finds the closest positive charging station to the drone which the drone has yet not visited and then assigns this station as the *next_destination.* Once the drone has visited all the positive stations it will assign null. This method is used in every iteration of the while loop in the *strategy_** methods.
7. *public Direction direction_to_station(Station station) :* This method takes a station and returns a direction to this station. The station given will always be the one in *next_desitiantion.* If the station is null it will adopt a self-sustaining strategy to find a direction, so it doesn't violate any game rules. Otherwise the way that the direction is picked is by finding the angle to the station and then finding the direction that would correspond to this angle. It is computationally quicker than iterating over all directions and finding the closets one to the station.
   The direction is given by where the angle falls in the range of + or - 12.5 of the actual direction so for example East is 45 degrees so if the angle is between 57.5 and 32.5 then we return East.
8. *public Direction skip_negative(Direction direction, Station station):* This function takes a direction and a station and returns a direction. This function ensures that after finding the direction to the station, that if the drone goes in this direction it doesn't fall in range of a negative charging station.

public class SimulateStatefulDrone

This class is used to simulate the different strategies the stateful drone will take. This class Is used to separate the logic of the StatefulDrone and the simulation aspect of its tactic.

Variables

The variables are the ones that can create a copy of a drone given to it. It will also keep the number of moves you want to run in the simulations of the drone's flight. It will *keep DD, MM, YYYY, lat, longi, seed* which will be the same as the drone that you want to run simulations on. The latitude and longitude is the starting position of the drone and seed is the seed given initially. The types are the same as before. The moves is the number of moves you want to simulate the run for. We have one class variable which is the *drone_type* which is always "stateful" since only stateful drones can run simulation.

Constructor - *public SimulateStatefulDrone(String DD, String MM, String YYYY, double lat, double longi, double seed, String drone_type, int moves)*

All the variables will be assigned to the class variables as appropriate. We need to pass the drone_type of the drone this is to ensure that a user doesn't try to use this class to simulate a stateless drone, which isn't possible since it would the throw an Exception.

Methods

The methods in this class or relate to making copies of the drone given and the executions of simulations to find the strategy that returns the most coins.

1. *public StatefulDrone drone_copy()* : This method doesn't take any parameter, returns a StatefulDrone. With the class variables it can create copies of the drone given.
2. *public int simulate()* : This method doesn't take parameters but returns an integer which corresponds to the strategy the stateful drone should use in the *run().* It creates 3 copies of the drone runs each copy on a separate strategy and returns the number of the one that yields the highest amount of coins.
3. *public void simulate_startegy_*(StatefulDrone drone) **(1, 2 or 3) : Takes a StatefulDrone and runs *strategy_*(int moves)* on this drone.

public class Station

This is the station class which keeps the coins, power and position of each station in the map. The features that we are given come with more attributes like the id of the station or the color of the marker however It was decided that only these 3 attributes are the ones that are necessary for our game. The only method that this class has is the *update_station()* which updates the station when a drone visits it, (excluding getters and setters).

Variables

It only has 3 instance variables which are coins, power and position of the station. They are of type double, double and Position respectively.

Constructor - *public Station(Feature feature)*

There is only one constructor and it takes a feature. This is because from a feature we can extract all the information we need to make a station. We make use of the feature's method *getProperty("coins")* and *getProperty("power")* to get the coins and power of the station and to get the position we get the geometry of the feature and then we can extract he coordinates which correspond to the latitude and longitude of the station.

Methods

We only have one method which is the method that updates the station once we visit it with a drone (excluding getters and setters).

1. *public void update_station(double transfer_power, double transfer_coins) :* This function takes the transfer_power and transfer_coins this is the drones total coins and power, double and then updates the stations power and coins. If the station has negative coins and power, it ensures that the power goes up to 0 and if the drone doesn't have enough coins or power to get the net to 0, this station will still have some negative power and coins. If the station is a positive charging station, we set power and coins to 0.
2. *Getters and Setters :* We have getters and setters for all the instance variables, they follow the convention that the other classes have.

public enum Direction

This is an enumerate class which keeps the 16 possible directions the drone can go in. It is useful to keep them in this class for easy access within other classes. The possible 16 directions are: N NNE, NE, ENE, E, ESE, SE, SSE, S, SSW, SW, WSW, W, WNW, NW, NNW.

public class Map

This is the class for the map where the drone will play the game. This class is important for the drone, so it knows where the stations are in the map. As well as all the station power and coins since they all change depending on the map.

Variables

This class has no class variables. This is because every time we create a new map the source and the feature collection will be different.

The instance variables for the map are:

1. *fc* : this of type FeatureCollection, this is used to get all the features in the map before we see the path the drone takes. It is used in the method *createJsonFile().*
2. *list_features* : this is the ArrayList of features in the map. We need this list because we will add the LineString to this list of features which will be the path of the drone. This list will be the one used in the method *createJsonFile().*
3. *list_stations* : this is an ArrayList of Stations. We get this by converting the features into stations and creating a list of all the stations in the map, this will be used by the drone throughout the runs.
4. *DD, MM, YYYY* : the day, month and year of the map respectively, String type

Constructor - *public Map(String DD, String MM, String YYYY)*

This class only has one constructor. This is because we will need to establish a connection to the server and retrieve the map, this is all done in the constructor**.** The map assigns DD, MM, YYYY to the corresponding variables. Then it continues to establish a connection to retreat the map from the destination we are given. All maps will be under:

"http://homepages.inf.ed.ac.uk/stg/powergrab/"+YYYY+"/"+MM+"/"+DD+"/powergrabmap.geojson"

This will be where we establish the connection to get the map from. The String was converted into a URL and then establish the HttpURLConnection *conn* with it. This will allow us to get an InputStream which then will be converted into a String with the method *convert(stream, charset).* This resulting string will be called a *mapSource* which we will be able to get the FeatureCollection *fc* from.

Once we have the feature collection, we can get the Feature List and as a result we can extract the *list_stations* and *list_features* variables.

Methods

There are only 2 methods that the class has, one is for converting the InputStream into a string and the other to make the GeoJson file.

1. *public String convert(InputStream InputStream) :* This method takes an InputStream and returns a String. The InputStream in our case will be the stream that we get when we establish the connection with the URL, once we have this, we can extract the features from it.
2. *Public void createJsonFile(ArrayList<Positions> positions, String drone_type, String DD, String MM, String YYYY) :* It will take an ArrayList of Positions, the drone_type, DD,MM and YYYY and returns void. This method is the one that takes care of making the GeoJson file. The drone_type, DD, MM and YYYY are used for the String formatting of the name of the .geojson file: “drone_type-DD-MM-YYYY.geojson”. It returns the original Json but with an added feature (the LineString which is the flight path of the drone).

public class Position

This class keeps the longitude and latitude of the drones and stations in the map. It is used to also calculate distance to other stations. The angle from the current position to the station, check if the position is in the play area ,find the next position if it goes in a given Direction and check if a distance is in range of a station.

Variables

This class has class variables. These will always be the same for all positions and they save time when calculating new positions. The class variables are *r* and the variations of r times cosine or sine of the angles 22.5 , 45 and 67.5.

1. *r* : is the double 0.0003 and is the distance a drone can travel in any direction.
2. *r(a)(b)* : (a = cos, sin) and (b = 22_5,45,67_5) this is the value of r*cos(45) for example, there will be 6 combinations of a and b, type double.

The 2 instance variables are longitude and latitude of the position since this will always be unique to the Position.

1. *latitude* : this is a double and is the latitude of the position.
2. *longitude*: this is a double and is the longitude of the position.

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

There is only one constructor and it takes a longitude and latitude and assigns them to the variable's *latitude* and *longitude* respectively.

Methods

The methods are all non-static, except *inRange()*, this is because we can use this method in various context outside the position class. However, the others are static because they depend on the current position.

1. *public Position nextPosition(Direction direction)* :Given a direction it returns the new position that results from moving in this direction. It uses a case switch to check the direction given and then calculates the new direction using geometry by calculating the change in the latitude and in the longitude.
2. *public boolean inPlayArea()* : Doesn't take any parameters and returns a boolean, checks if the current position is in or out of the play area. That is if latitude is between 55.946233 and 5.942617 and longitude is between -3.184319 and -3.192473 then it returns True, otherwise false.
3. *public double distanceStation(Station station) :* This takes a Station and returns a double the distance from the current position and a station. It returns the Euclidean distance between the current position and the stations position.
4. *public double angleStation(Station station) :* Takes a Station and returns a double which is the angle between the station and the position. This makes use of trigonometry rules by centering the position and finding the station with respect to the position, it first locates in which quadrant the station falls in the unit circle and then finds the angle accordingly by making use of the angle calculated by arctan(opposite/adjacent).
5. *public static boolean inRange(double distance) :* Given a distance, a double, returns a boolean. This function returns True if the distance given is less than or equal to 0.00025, the distance used in this function is found by using the *distanceStation(Station station)* method.
6. *Getters and Setters :* We have getters and setters for: latitude and latitude. They are defined by the following convention: get or set + name of variable with capital first letter. For example, the getter and setter for coins would be *getLatitude()* and *setLatitude (double latitude).*

public class App

This is the app class which will run the simulation framework for the drone. It takes 7 command line arguments: day, month, year of the map, starting latitude and longitude of the drone, a seed and the drone type. Given this information the app class will make the corresponding drone in the corresponding map. The logic is all done within the *main()* method (the only method in this class).

Public static void main(String args) throws IOException

First it parses the 7 command line arguments into variables, they are given in this order: DD, MM, YYYY, latitude, longitude, seed, and drone type. Then uses a simple if statement which separates the 2 cases stateless and stateful drones. Creates the corresponding drone object. Then uses the *run(int moves)* method to run the drone. Finally creates a FileCreator object which takes the map and the drone after the run and returns the 2 corresponding files.

public class FileCreator

This is the class that is in charge of making the text and JSON file. It will take a map and a drone and will produce the files after the drone has moved around the map.

Variables

This class has no class variables, this is because the files depend on the drone and map that is given to this class. There are only 2 variables in this class, the drone and the map of the drone.

1. *drone* : this is the drone that we want the files on, the type is Drone so can be StatefulDrone or StatelessDrone.
2. *map* : this is the map that the drone played in after it has finished. It is of type Map.

Constructors - *public FileCreator(Stateless drone , Map map) , public FileCreator(StatefulDrone drone , Map map)*

This class has 2 constructors to take care of both cases for the 2 different type of drones, they both take a drone and a map and assign these values to drone and map respectively.

Methods

All the methods are non-static this is because they need a drone and a map to work. They take the drone and the map from the instance variables.

1. *public void createTxtFile_movements() :* This method doesn't take any inputs and returns void. However, the method produces the .txt file by using a *PrintWriter* object (*return_txt*). Each line of the file is defined by each entry in the *movements* variable of the drone. So, the function iterates over this list and prints them out to the text file line for line. It throws an IOException.
2. *public void createJsonFile_Flight() :* The method doesn't take any inputs and returns void; this is because we are creating the GeoJson file. It creates the JSON file by calling the *map* method *createJsonFile()* this method takes the drones: *positions, drone_type* and *DD,MM,YYY* of the map. This method is explained in the Map class.

3 - Stateful drone strategy

Problem

The goal of our drone is to get as many coins as possible in a map. This consists in go to as many positive charging stations and avoiding the negative ones. Since the drone has limited amount of power it needs to make sure to not run out of power when it moves. Our problem is a “travelling salesman problem”, this is because we have a list of stations and we want to find a path that is the shortest to visit all the stations that are positive. We want to make the path the smallest because we have limited amount of power and we want to only go to the positive stations because we want to maximize our coins. So, we want to maximize our coins subject to power.

What the drone remembers

The drone has various variables that it keeps track of in order to improve its score. Firstly, the drone keeps track of what station it must go to. Once the drone finds the closest positive station from its current position it will head towards this station and once it arrives it finds the next station to go to. This ensures that the drone visits the station it is set out to visit. This is the main advantage that the stateful drone has over the stateless one. This is because it knows in what direction it must go and get to a station quickest. The stateless drone does not know where to go, it will go in random directions and then hope to find a direction that falls in range of a positive charging station, the process of moving in random directions and avoiding negative ones is not efficient, this is where the stateless drones strategy is lacking.

The drone also keeps track of all the directions that the drone flies in. This list of directions is vital for the drone to know if it is stuck or not. This is because if the drone detects that it is repeating the same 2 directions for more than 10 times the drone is stuck and so it will decide to go through the negative station which is causing it to get stuck. Even though it might result in the loss of coins it will allow the drone to visit all the other positive stations that it hasn’t visited yet.

Furthermore, the drone keeps a list of all the positive stations it should avoid, these are the ones that fall in the range of a negative station. However not all the strategies that the drone undertakes creates this list, in that case the list will be empty. In comparison with the stateless drone that doesn’t this helps the drone get better scores because it can see in advance that it could get stuck or visiting a power station can result in loosing coins.

Finally, the drone keeps a list of all the stations of the map. This allows the drone to know exactly where all the stations are (negative and positive). This gives the drone a huge advantage in getting better scores because it enables it to only go to stations with positive coins and knows where it should not go since it knows where all the stations are in the map.

Strategy

The strategy used by the stateful drone is a greedy strategy. There are 3 sub strategies that the drone takes, they all, are greedy strategies. The stateful drone will run simulations of itself with the 3 strategies on the map and then picks the strategy which maximizes the amount of coins. This one will be used by the drone in the flight.

The first strategy takes a list of all the charging stations and goes to all the positive ones, it goes to the closest one using the find_station() method. Then it finds the general direction to this station using direction_to_station() from its current position and ensures that this direction doesn’t fall in range of a negative charging station, if it doesn’t it goes to the general direction otherwise it finds a new direction that is also close to the station but doesn’t fall in the range of a negative charging station this uses the skip negative() method which also considers if the drone doesn’t leave the play area and does this until it has visited all the positive stations.

The second strategy is like the first tactic however the key difference is that before it flies it scans the map, this is done through the stations_to_aviod() method before the drone starts flying. Goes through the list of all the stations and choose to not visit the positive stations which are in the range of a negative power station. This means that if a positive station falls in range of another station the drone will choose to avoid these stations because they are too close to the negative

stations. Then the drone goes to all the positive stations except the ones it filtered out when it went through the list of stations just like in strategy one.

The third strategy involves taking on negative stations if the drone gets stuck. The drone will again, like in the first strategy, go to all positive charging stations in order of closest distance to the drone. However, there can be the case where the drone can´t find a direction to go in because a station is in range of a negative station or if stations are very close together thus, it gets stuck. The strategy will check if the drone is stuck if it is it will get a random direction to go in. This is implemented in the is_stuck() method which ensures that if it gets stuck it goes in a random direction though it might result in negative loss of power. Then the drone will continue as usual visiting all the positive power stations. Once it visits all the stations it will try to go back to the stations it had missed and see if it can pick them up.

Once it has visited all the positive power stations the drone will adopt a safe holding pattern that will ensure that the drone doesn't visit any negative charging stations and doesn't leave the map. The drone will pick the north direction and check if the 2 conditions hold if it doesn't it will pick the next direction in the clockwise direction (NNE) and check again, it will do this recursively. This is because the *next_destination* variable will be null if it has visited all the positive power stations.

Comparison with stateless drone

We can see that the path of the stateful drone has purpose and once it visits all the positive stations it adopts a safe holding pattern to avoid all the positive stations. This can be seen in the top left corner where the drone is going north and south continuously (next to 2 negative stations). For this map the stateful drone gets 100% of all the possible coins using its strategy (1880.44 total coins).

For the stateless drone we can see an almost "random" path. However, we do see that this drone does avoid negative stations and with its one look ahead sometimes manages to land in range of a positive and pick up some coins and power. In this map the stateless drone only achieves 554.64 coins. This is only 29.5% of the possible coins on the map. This is performance of the drone is what would have been expected since the stateless drone is blind compared to the stateful drone.

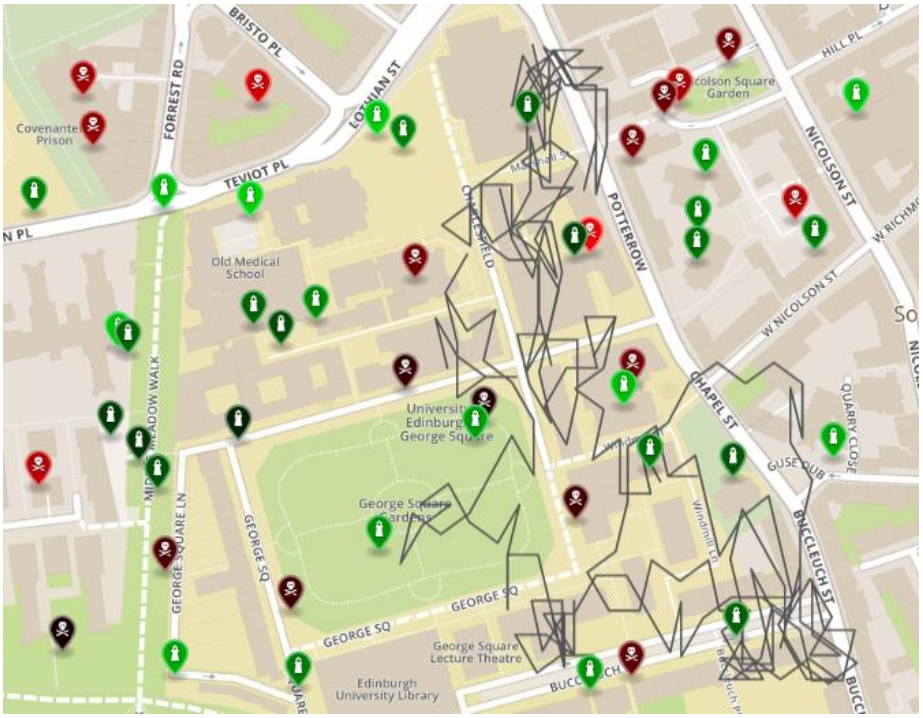


Fig 1 : Stateless drones flight path on 01-01-2019

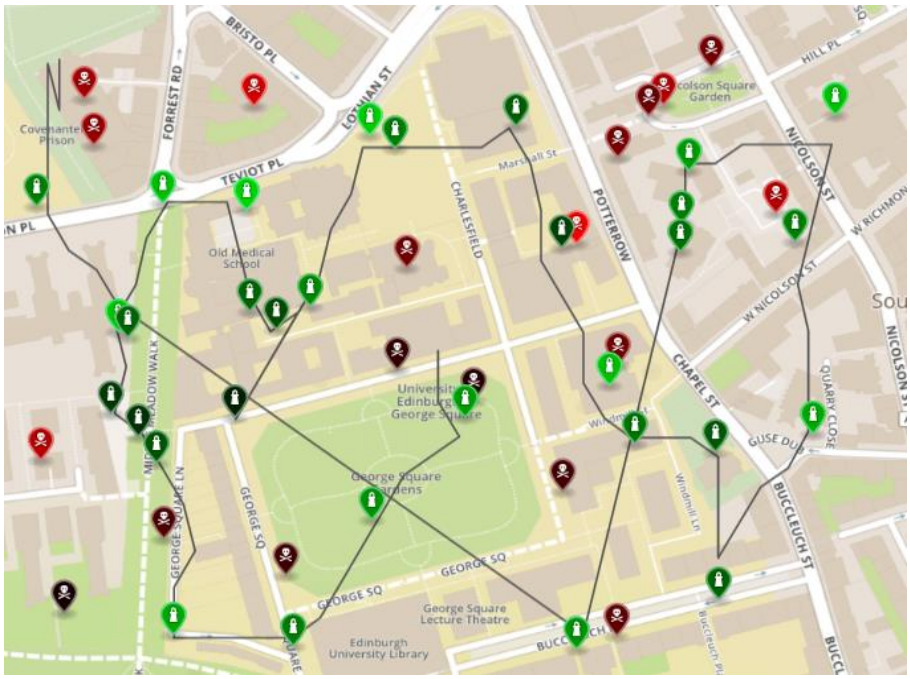


Fig 2 : Stateful drones flight path on map 01-01-2019

Results

After using Bora´s PowerGrab evaluator. We got results for the stateful drone over all the maps form 2019 and 2020. The minimum ratio (total coins collected over total coins possible in a map) of coins collected was 83.18% the median ratio was 100% and the average ratio was 98.87%. This showed us that our drone strategy achieved perfect scores for more than half of the maps in the game. And its average time to run was 0.67 seconds.

Conclusion

Overall the stateful drone performed well under the conditions of the game, achieving 98.87% average for picking up coins. Furthermore, the average runtime for each map was 0.67 seconds given the fact that it does 3 strategies and picks the best is relatively quick.

However, if the rules change, for example the drone must take less moves or the drone losses more power per move, this means that our strategy will need to change. This is because greedy algorithms might fail to find the global optimum solution, they usually find paths 25% longer than shortest possible path(1.) as a result with harder rules the drone might not be able to visit all the stations because it could lose too much power if it doesn't find the shortest rule.

Other algorithms can be used to solve the traveling salesman problem applied to our game. That is visiting all the positive charging stations in the smallest distance possible. These involve heuristic algorithms that usually find good solutions 2-3% from optimum solution (2.).We could have applied one of these algorithms to further optimize the total travel distance the drone takes.

Also the fact that our drone takes a random direction to solve the situation when it gets stuck it is not ideal, preferably we would like to have tactic which finds a direction or a combination of directions to get to the positive station while not getting in range of a negative charging station or getting stuck between positive stations.

References

1. Johnson, D.S. and McGeoch, L.A., 1997. The traveling salesman problem: A case study in local optimization. *Local search in combinatorial optimization*, *1*(1), pp.215-310.
2. Rego, C., Gamboa, D., Glover, F. and Osterman, C., 2011. Traveling salesman problem heuristics: Leading methods, implementations and latest advances. *European Journal of Operational Research*, *211*(3), pp.427-441.