Informatics Large Practical - PowerGrab Report

Greig Huth s1532620

Contents

Chapter 1

Software Architecture

1.1 Overview of PowerGrab

Within my implementation of PowerGrab i decided i would only need 7 classes: App, Direction, Position, Drone, StatefulDrone, StatelessDrone and Station.

App class is simply the main class of the program, something like this is of course needed in order to actually run PowerGrab and App was the default name for it in Maven.

The decision to make Direction an enumerator was one i decided early on. It can only take 16 different constant values, and enumerators are a good way to implement classes with fixed values at compile time.

When it came to implementing the drone, i decided it would be best to have a Drone super-class and have the stateless and stateful drones be sub-classes. This is because i knew there would be methods they would have to share, so to avoid duplicating code, i thought this would be the best strategy. Initially i was considering using an interface, as this would remove the need for explicit class casts within the main function, but this would have resulted in duplicated code in the classes that implement the interface, and with the way my main is structured, i check the instance of the drone anyway, so it made sense to use a class hierarchy architecture rather than interfaces.

I decided to make Station its own class for three reasons. The first is that it made accessing the stations far less clunky than using the map-box API. At the beginning of the code i unpack all the JSON "features" into an array of Stations to achieve this. Secondly, it allowed me to create a score method in order to calculate a score for all the stations, this is used in the stateless drones decision making. Finally, it allowed me to easily edit a copy of the stations rather than the original collection. They need to be edited when the drone charges from them, as when the drone charges it drains the station of its coins and power, but the original needed to be preserved when generating the output JSON file, so this solution seemed to fit quite naturally.

I chose to have the power and coins outside of the drone class. This is because in my implementation of the stateful drone, the amount of power and coins it has never factors into any of its decision making, additionally part of the specification of the stateless drone was it to not be able to keep track of its current power and coins, so in the end this seemed to be the best way of doing it. In addition, i chose to store the coins and power within my program as doubles for the simple reason that in the JSON file we are given, coins and power have precision to 16 decimal places, which is the same as a double, so it made sense to store them to their intended precision.

Chapter 2

Class Documentation

The following section contains brief descriptions for all the classes used in the practical. "Getters" and "setters" have been omitted from the documentation as their functionality is trivial, and their inclusion in this document would serve only to clutter it.

2.1 App

This is the class that contains the main function that all the other parts of the practical are run in.

Methods

public static void main(String args[ ])
This is the main class which the rest of the code is controlled and run from. Firstly it parses out the arguments and then initialises the drone based on these arguments. Which version of the drone it runs is specified by one of these arguments. After all the initialisaion is completed, the main game loop is run. The game will keep going on while the number of moves left is greater than 0 and while the amount of power left is greater than or equal to 1.25(since it needs at least 1.25 power to move). Each loop through the game is specified as follows: the next direction the drone should move in is calculated, then the drones position is updated with the direction to move in. It is at this point that power and moves variables are updated. After this, the game checks to see if the drone is in range of any stations, if it is then it charges from them, if it isn't it moves on. In the last part of the loop, the output files are updated with the drones new move information. After the game has finished, the output files are created and written to.

public static ArrayList<Station> updateMap(Station charge,ArrayList<Station> allStations)
Returns a new map with the given "charge" stations coins and power attributes drained.

public static Station getChargeStation(Position dronePos, ArrayList<Station> allStations)
Returns the closest station to the drone, that the drone is also in range of, so the drone can charge from it.

public static double distance(Position pos1, Position pos2)
Returns the Pythagorean distance between the two given positions.

public static ArrayList<Station> unpackFeatures(List<Features> features)
Extracts the stations from the geo-json string, and gives each its own Station instance, then returns them all as an array.

public static Position unpackPosition(Feature feature)
Extracts the latitude and longitude from the given feature and returns a Position instance with these attributes.

public static Feature makeLine(Position currPos, Position nextPos)
Creates a feature containing a line that represents the drones flight path after a move.

public static Drone initDrone(Position start,int seed,String v,ArrayList<Station> allStations)
Returns an initialsed drone based on if the game is running a stateless or stateful version.

public static String getMapSource(String d, String m, String y)
Gets the geo-json from the web-server, the parameters represent the day, month, and year which comprise the name of the file, this is different depending on the command line arguments.

2.2 Direction

This enumerator represents the 16 different directions the drone is able to move in, for a complete list of the directions, see Figure 3 in appendix A.
Each constant in the enumerator is comprised of of the values to be added to the latitude and longitude to successfully move in the corresponding direction. The class uses Pythagorean distance to calculate how far the drone should move in each direction. For the formula used see Figure 4 in Appendix A.

Attributes

private double width - value used in calculating the next position.
private double height - value used in calculating the next position.

Methods

Direction(double w, double h)
Enumerator constructor, not explicitly called. Used by the enumerator to get the width and height values for each direction.

2.3 Position

This class is used to represent the position of the drone and all the stations within PowerGrab.
Attributes

private double latitude - the latitudinal component of the position.
private double longitude - the longitudinal component of the position.

Methods

public Position(double latitude, double longitude)
Constructor method.

public Position nextPosition(Direction direction)
Returns the next position to move to in based on the given direction.

public boolean inPlayArea()
Decides whether or not the instances' position is within the play area. Returns true if it is and false otherwise.

public boolean equals(Position pos)
checks to see if the given position is equal to the instance one. Returns true if they are equal, false if they are not.

2.4 Station

This class is used to represent the charging stations in the game.

Attributes

private string id - The id of the station.
private Position position - The position of the station.
private double coins - The number of coins the station holds.
private double power - How much power the station holds.
private string marker - the marker of the station, either "danger" or "lighthouse"

Methods

public Station(String id, Position position, double coins, double power, String marker)
Constructor method.

public double getScore()
Returns the "score" of the station by summing its *coins* and *power* values.

2.5 Drone

This class is used to represent the drone that will play the game.

Subclasses:
StatelessDrone, StatefulDrone.

Attributes

private Position position - The current position of the drone.

Methods

public Drone(Position position, int seed)
Constructor method. The *rnd* attribute is also initialised in this method.

public ArrayList<Direction> legalDirections()
Uses the *directions* attribute and returns a list of directions the drone can move in that result in legal positions (positions that are in the play area).

public Direction closestDirection(ArrayList<Direction> legalDirections, Station station)
Returns the direction, from the given legal directions, that moves the drone closest to the given station.

2.6 Stateless Drone

This class is used to represent the Stateless drone.
Extends:
Drone

Attributes
private final Random rnd - Random number generator.

Methods

public StatelessDrone(Position position, int seed)
Constructor method. Passes position parameter to Drone constructor. Initialises rnd with the given seed.

public Direction calculateDirection(ArrayList<Station> stations)
Returns the best direction to move in. It first filters out illegal moves and then finds any good stations within range of the drone after any of its possible moves. If there are not any stations in range then it moves in a random direction. If there are any stations in range, it finds the one with the largest score and then finds the direction that would move the drone closest to it. It then returns the chosen direction.

private Direction randomDirection(ArrayList<Direction> legalDirections)
Picks a random direction to move in. Uses a seed so the randomness is repeatable.

private ArrayList<Station> stationsInRange(List<Station> stations, List<Direction> directions)
Returns all of the stations that the drone is in range of after any of its possible moves.

2.7 Stateful Drone

This class represents the Stateful drone. For a more complete explanation of this class, see chapter 3.

Extends:
Drone

Attributes

private ArrayList<Position> alreadyVisited - List of positions the drone has already been to. *private ArrayList<Station>* goodStations - a list of all the stations with the "lighthouse" marker.
private ArrayList<Station> badStations - a list of all the stations with the "danger" marker.
private ArrayList<Station> orderedStations - an ordered list of all the good stations. Order of the stations is dictated by which order the drone will visit them. It visits the one that is closest to it at that time.

Methods

public StatefulDrone(Position position, ArrayList<Station> stations)
Constructor method. Passes position to superclass. Uses stations to populate the goodStations, badStations and orderedStations attributes.

public Direction decideDirection(Station destination)
Decides what direction to move in. It first filters out illegal moves and orders the different moves it can make by their distance to the destination. For each of these ordered moves, starting with the closest to the destination, it checks to see if the move will result in being in danger. If it does then it picks the next direction in the order and try's that. It does this until it finds a direction that is safe to move in. If it doesnt find any safe directions, the it picks the closest direction.

private void seperateStations(ArrayList<Station> stations)
Seperates stations into good stations and bad stations, depending on the marker. The "lighthouse" marker means good station and the "danger" marker means its a bad station.

private void sortStations()
Orders the good stations, to decide an order in which to be visited. It chooses this order based on what station is currently closest to the drone at that time.

private ArrayList<Direction> sortDirections(ArrayList<Direction> directions, Station destination)
Returns a sorted list of all the directions given. It is sorted in ascending order of the distance between the drones position after moving in a direction, and the destination station.

private boolean inDanger(Direction chosenDirection, Station destination)
Decides if the drone is in danger of losing power and coins. If the drone is in range of a bad station and a good station and the good station is closer then it returns false. If it is in range of only a bad station it returns true.

Chapter 3

Stateful Drone Strategy

3.1 Basic Explanation

The basic explanation of how the stateful drone is as follows. When the drone is initialised, before the main game loop has even begun, it decides what order to visit all the good stations in. This order is based on the current position of the station the drone will have just visited. For example: If the drone were to start at position (1,1) and the nearest station was at (2,2) then this will be the first station the drone visits. Then it bases its next destination what station is closest to the station it just visited. So it would find the station closest to the coordinate (2,2). It does this until it has an ordered list of all the good stations the drone needs to visit. Then the main game loop can begin and the drone then needs to decide on what directions to move in.

At the beginning of each move, the drone is given a station to move to, called the destination, which will be the first element in the list described above. It picks the direction to move in by ranking all the possible moves it can make from its current position in terms of how close they are to its destination. Firstly the drone picks the direction closest to the destination and tries that. If this move results in being in range of a dangerous station then it will try the second best direction, and so on and so forth until it finds a move. If all the moves result in being in danger, then the drone will just pick the original shortest one. If the move it picks results in being in range of both a dangerous station and a good station, but its closer to the good station, it will pick that move. Because the drone always charges from the closest station, and since the distance the drone can move is larger than the stations area of effect it will always be able to move to a good station, charge from it and then move out of range of a dangerous station before it then charges from it. If the move it picks leads to a position the drone it has already been to (i.e is it getting stuck) it will ignore that direction and try the next one.

After the stateful drone has exhausted the list of stations to visit, it will then make its way to the original starting position and stay in the area of it. This is to ensure that the drone does not stray unexpectedly into range of a danger station. It does this for the remainder of the moves left in the game.

One of the key improvements i made over to the stateful drone over the stateless drone was to ensure there is no randomness in its implementation, randomness leads to uncertainty and this in not conducive to getting a consistently high score. I also decided that it would be a good improvement to allow the stateful drone to keep track of what stations it needs to visit and what ones it has already visited and also what positions the drone has already been to. This allows the drone intelligently navigate the map, making sure to not waste time as it is able to go to the next station as quickly as possible, while also avoiding danger stations along the way and avoid getting stuck in infinite loops.
After a move has been completed, the game then checks if the drone is in range of any station, and charges from them if it is. If the station it is in range of is the destination station, then it removes it from the list of station to visit, so it doesn't try to keep going to the same one forever.

The flight path of both drones on the same map are shown in figures 1 and 2 on the next page.

3.2 Diagrams

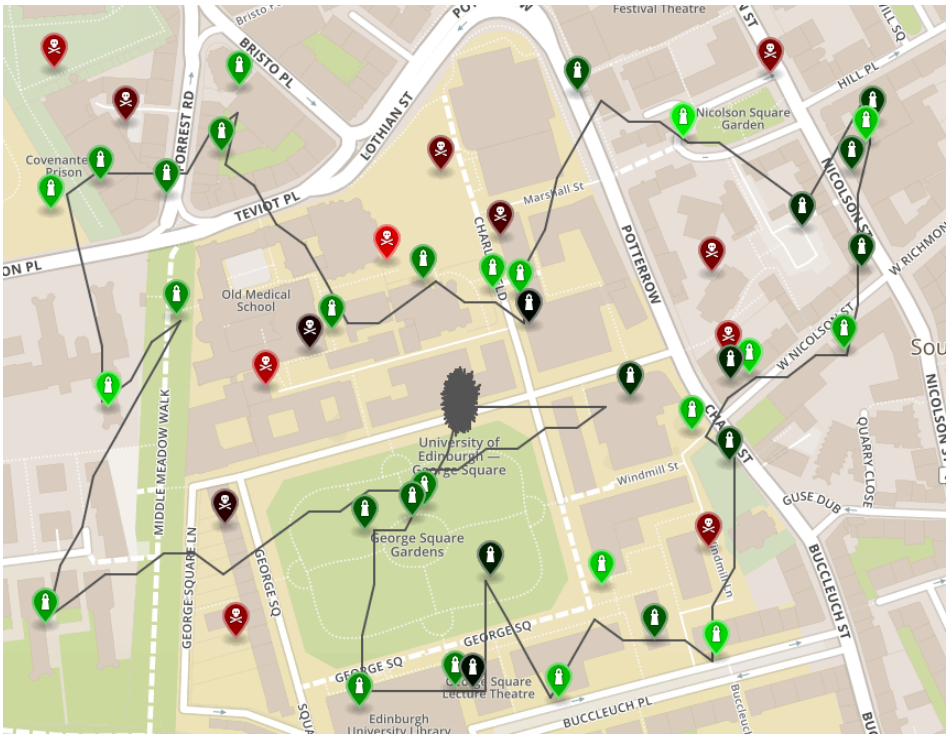

Figure 1: Stateful drone flight path on the map for 12/12/2019.

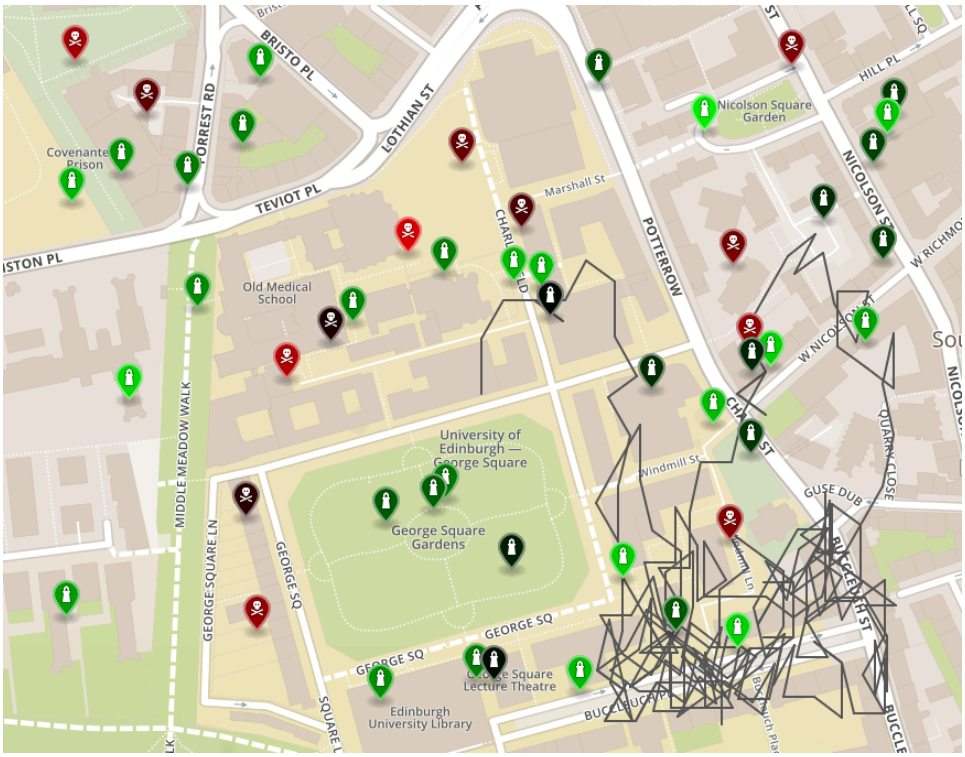

Figure 2: Stateless drone flight path on the map for 12/12/2019.

Chapter 4

References and Appendix

4.1 References

https://www.javatuples.org/
https://docs.mapbox.com/android/java/overview/
https://commons.apache.org/proper/commons-io/

4.2 Appendix

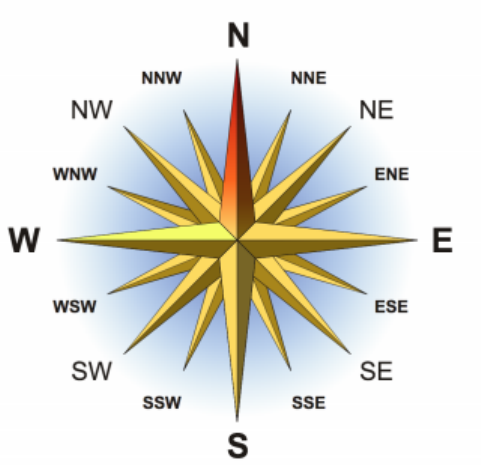

Figure 3: Compass used as reference for the directions the drone can move in.

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

Figure 4: Formula for pythagorean distance.