Informatics Large Practical

Project Report

Dimitris Christodoulou (s1738739)

Table of Contents

1 Software architecture description

To create a coherent structure for the application we need a class for each important thing in our problem domain[1]. According to the single responsibility principle[2] every class we identify should have responsibility only over a single part of the functionality of the application.

The game is location-based with specific geographical boundaries. Therefore, we use the **`Position`** class to encapsulate the coordinates (latitude, longitude) and provide methods for ensuring that some position is within the boundaries of the game and to calculate the resulting positions after performing some move.

The game contains stations. Hence, a **`Station`** class is needed to encapsulate all the relevant fields (position, power, coins) and the methods related to a station.

The entity fundamental to the application is the autonomous drone which moves around the map and collects power and coins from the stations. Therefore, a **`Drone`** class is needed to encapsulate its fields (power, coins, position). There are two types of drones: The stateless and stateful drones. These two types of drones, for the most part share the same functionality with the exception of the algorithm that determines which moves they perform to collect the coins. Thus, `Drone` is implemented as an abstract class with the methods for the common functionality defined and the functionality that has different implementation for each type of drone declared as abstract methods.

To implement the functionality specific to each drone type (path planning) we need the **`StatelessDrone`** and **`StatefulDrone`** classes which extend the abstract `Drone` class. Each of these implements the abstract methods of the `Drone` class. As `Drone` is an abstract class, only `StatelessDrone` and `StatefulDrone` can be instantiated.

Part of the stateful drone's path finding algorithm uses the A* path-finding algorithm. Positions in A* are represented as nodes, which are associated with some scores. For this reason the **`Node`** class is needed which is implemented as a private nested static class of the `StatefulDrone` class. This class is only needed for the path finding algorithm of the stateful drone. Hence, implementing it as an inner class increases encapsulation since it is only useful for one class and makes it more maintenable and readable by placing it is closer to where it is used.

The drone's path is a sequence of discrete moves. Therefore, the **`Move`** class is needed to store the details of each move (position before/after, coins and power after the move etc), make it possible to elegantly store the move history of the drone, and for the stateful drone's algorithm to be able to create a sequence of moves to be executed later by creating a list of `Move` objects.

The drone can only move in 16 specific directions. To define those directions we use the **`Direction`** enum class which associates each direction with a move angle.

The maps showing the locations of the charging stations are available from the web server as Geo-JSON documents. The **`GeoJSON`** class is responsible for all interactions with the Geo-JSON document including reading the document from the server, parsing it and adding the drone's path to it. This abstracts away all the details of how Geo-JSON works from the rest of the application.

To represent the map of the game, we use the **`GameMap`** class which stores the `GeoJSON` object associated with it and keeps a Set of the stations that are on the map, which are extracted from the Geo-JSON document. It also provides methods that can be used to query the game map (e.g. finding the nearest positive station from a position) and interact with the GeoJSON object (e.g. adding the path to the document).

The game has rules. Examples of such rules are the distance covered by every move, which the boundaries of the map are and how much power the drone has initially. These rules are stored as static final variables in the **GameRules** class. This allows easy access of the rules of the game throughout the application and easy modification of the rules as they are all in one place.

Finally, the **App** class contains the main method which is the entry point to the application. It handles the inputs, runs the simulation and outputs the results to files.

2 Class documentation

2.1 GameMap

The GameMap class represents the map of the game for a specified date. The map consists of stations which are extracted from the Geo-JSON document for the specified date (retrieved from the web server). It provides methods for extracting information from the game map, such as finding the station a given position is in the range of. It also provides methods that interact with the Geo-JSON document such as plotting the drone's path to the document and outputting the Geo-JSON document to a file.

Constructor and Description
public GameMap(String year, String month, String day) Allocates a GameMap object and initialises it to represent the map with the given date. Uses the given date (represented as Strings) to create a new GeoJSON object and from that it extracts the set of stations.

Fields

Modifier and Type	Field	Description
private final GeoJSON	**geoJsonDocument**	The Geo-JSON document representing the map
private final Set<Station>	**stations**	The set of stations present on the map
private final double	**perfectScore**	The perfect score a drone can achieve in this game map. It is equal to the sum of the coins of the positive stations on the map.

Methods

Modifier and Type	Method and Description
public Station	**getStationToConnect(Position pos)** Returns the station the given position is in the range of (i.e. the station a drone at the given position can connect to). If the position is not in the range of any station, null is returned.

public Station	**getNearestPositiveStation(Position pos, Set<Station> excludedStations)** Returns the nearest station with a positive amount of coins/power, excluding the stations in the excludedStations set and the station the given position is in the range of. If no such station can be found, null is returned.
public void	**outputMapToFile(String fileName)** Outputs the Geo-JSON document to a file with the given file name.
public void	**addDronePathToMap(Drone drone)** Adds the path of the given drone to the Geo-JSON document.

2.2 Position

The `Position` class represents positions on the map, defined by their coordinates (latitude, longitude). It includes methods for calculating the resulting position after moving in a certain direction, for calculating the distance relative to other positions and for determining whether a position is within the predefined play area.

Constructor and Description
public Position(double latitude, double longitude) Allocates a `Position` object and initialises it to represent the given coordinates.

Fields

Modifier and Type	Field	Description
public final double	**latitude**	The latitude of this position
public final double	**longitude**	The longitude of this position

Methods

Modifier and Type	Method and Description
public Position	**nextPosition(Direction direction)** Returns the resulting position after a move to the specified direction. The move distance is fixed and accessed from the `GameRules` class.
public boolean	**inPlayArea()** Returns whether the position represented by this `Position` object is within the play area boundaries. The boundaries are fixed and accessed from the `GameRules` class.
public double	**distance(Position pos)** Returns the euclidean distance between this position and pos.
public boolean	**equals(Object o)** Compares two positions for equality. Two positions are equal if and only if they have the same coordinates.
public String	**toString()** Returns a string representation of this position which consists of the relevant latitude and longitude.

public int	**hashCode()** Returns a hash code for this position.

2.3 Direction

`Direction` is an enum class representing the 16 directions the drone is allowed to move.

Enum Constants:

N, NNE, NE, ENE, E, ESE, SE, SSE, S, SSW, SW, SSW, SW, WSW, W, NWN, NW, NNW

Constructor and Description
public Direction(double angle) Allocates a `Direction` object and initialises it to represent the given angle. The angles start at the North with 0 and increase clockwise.

Fields

Modifier and Type	Field	Description
public final double	**angle**	The angle represented by this direction.
public final double	**sine**	The sine of the angle of this direction.
public final double	**cosine**	The cosine of the angle of this direction.

Methods

Modifier and Type	Method and Description
public Direction	**getOppositeDirection()** Returns the opposite to this direction.

2.4 Move

The `Move` class represents a move of a drone. It contains all the properties of a move such as start/end position, direction of move etc. When a `Move` object is created, it has not been performed by the drone. When the object is created only the move direction and the relevant drone are specified. The other fields are populated after the move has been executed. The `Move` class provides a method to execute a move and a method to output a String representation of a move.

Constructor and Description
public Move(Drone droneToMove, Direction moveDirection) Allocates a `Move` object and initialises it to represent the move of the specified drone to the specified direction.

Fields

Modifier and Type	Field	Description
private Position	**positionBefore**	The position of the drone immediately before it performed the move.
private final Direction	**moveDirection**	The direction of the move.

private Position	**positionBefore**	The position of the drone immediately after it performed the move.
private Double	**coinsAfter**	The coins the drone had immediately after it performed the move.
private Double	**powerAfter**	The power the drone had immediately after it performed the move.
private final Drone	**droneToMove**	The drone that is to perform (or has performed) the move.

Methods

Modifier and Type	Method and Description
public void	**executeMove()** Executes the move. Stores in the fields of this Move object the position of the drone before/after and its coins & power after the move.
public String	**toString()** Returns a String representation of this move that includes all the fields with the exception of droneToMove

2.5 Station

The `Station` class represents a station that is described by its position, coins and power.

Constructor and Description
public Station(Position position, double coins, double power) Allocates a `Station` object and initialises it to represent the station with the specified position, coins and power.

Fields

Modifier and Type	Field	Description
private final Position	**position**	The position of the station
private double	**coins**	The amount of coins the station has. Can be positive, negative or zero.
private double	**power**	The amount of power the station has. Can be positive, negative or zero.

Methods

Modifier and Type	Method and Description
public double	**distanceFromPosition(Position pos)** Calculates and returns the euclidean distance from the given position to the position of this station.

`public boolean`	**`positionInRange(Position pos)`** Returns whether the given position is within the range of the Station. The range of the station is accessed from the `GameRules` class (`CONNECT_DISTANCE`).
`public void`	**`connect(Drone drone)`** Connects the specified drone with this station. Coins and power are transferred between the drone and this station.
`public boolean`	**`isPositive()`** Returns whether this station is positive (i.e. has positive coins or power).
`public boolean`	**`isNegative()`** Returns whether this station is negative (i.e. has negative coins or power).

2.6 GeoJSON

The `GeoJSON` class represents the Geo-JSON document from which the game map is obtained. Each Geo-JSON document represents a feature collection, which is what the `GeoJSON` class stores. It provides methods for retrieving the document from the web server, parsing the stations from the document, saving the document to a file and plotting a path to the document. The constructor is private. GeoJSON objects can be obtained through the `retrieveDocumentForDate()` factory method.

Constructor and Description
`private GeoJSON(FeatureCollection featureCollection)` Allocates a `GeoJSON` object and initialises it to represent the given feature collection.

Fields

Modifier and Type	Field	Description
`private final FeatureCollection`	`featureCollection`	The feature collection represented by the Geo-JSON document.

Methods

Modifier and Type	Method and Description
`public static GeoJSON`	**`retrieveDocumentForDate(String year, String month, String day)`** Connects to the web server, retrieves the Geo-JSON document for the specified date and extracts the feature collection represented from the Geo-JSON document. A a new GeoJSON object for this feature collection is returned.
`public void`	**`addPathToGeoJSON(List<Position> pathPositions)`** Adds the given path to the feature collection of this Geo-JSON document.
`public Set<Station>`	**`getStationsFromMap()`** Returns the set of the stations contained by this Geo-JSON document.
`public String`	**`toString()`** Returns a String representation of the feature collection.

2.7 Drone

The Drone abstract class represents a drone within our game. This class provides implementations of some drone operations that are the same irrespective of the drone type. A class that extends the Drone class must implement the **findPath()** and **getDroneType()** methods.

Constructor and Description
public Drone(Position position, GameMap gameMap, long randomSeed) Allocates a Drone object and initialises it to represent the drone with the given position, interacting with the given game map. A random seed is also provided to ensure reproducibility of the results.

Fields

Modifier and Type	Field	Description
protected Position	**position**	The position of the drone.
protected double	**coins**	The amount of coins held by drone.
protected double	**power**	The amount of power held by the drone. The initial value is accessed from the GameRules class.
protected final Random	**rnd**	The random number generator used by the methods of the drone, that uses the random seed passed to the constructor.
protected final List<Move>	**moveHistory**	The sequence of moves performed by the drone.
protected final GameMap	**gameMap**	The game map the drone interacts with.

Concrete Methods

Modifier and Type	Method and Description
public void	**move(Direction d)** Moves the drone to the specified direction. Its position is appropriately updated and the drone connects to the station within range (if it exists). The power of the drone is reduced by the amount of power consumed by one move.
public double	**addOrSubtrCoins(double deltaCoins)** Adds or subtracts the given amount of coins to/from this drone's coins (depending on the sign of deltaCoins). The value returned is the residual amount of coins that could not be added/subtracted. If coins are subtracted and the drone does not have enough coins to lose, it loses all its coins and the amount of coins that is left is returned. In all other cases, the given amount of coins is added/subtracted and 0 is returned.
public double	**addOrSubtrPower(double deltaPower)** Adds or subtracts the given amount of power to/from this drone's power (depending on the sign of deltaPower). The value returned is the residual amount of power that could not be added/subtracted. If power is subtracted and the drone has less power than the amount to be subtracted, it loses all its power and the amount of power that is left is returned. In all other cases, the given

	power is added/subtracted and 0 is returned.
public List<Position>	**getPath()** Returns a list of the sequence of positions the drone has visited. Extracted from the drone's move history.
public void	**outputMoveHistoryToFile(String fileName)** Saves the drone's move history to a text file with the given filename.

Abstract Methods

Modifier and Type	Method and Description
public abstract void	**findPath()** Finds the path the drone follows, dictated by the specific drone type's strategy. Stores the moves performed in the drone's move history.
public abstract String	**getDroneType()** Returns as a String the name of the type of the drone.

2.8 StatelessDrone

The `StatelessDrone` class extends the `Drone` abstract class. It represents a drone within our game that is stateless. This means that it has no local state, it is memoryless and its decision on the next move can only be based on information about the sixteen positions that are reachable from the drone's current position.

Constructor and Description
public StatelessDrone(Position position, GameMap gameMap, long randomSeed) Constructs a StatelessDrone object and invokes the constructor of the Drone superclass.

Methods

Modifier and Type	Method and Description
public void	**findPath()** Finds the path to be followed by the stateless drone. This method calls the nextMove() method continuously to determine each move until the maximum number of moves has been performed or the stateless drone does not have sufficient power for further moves.
private Move	**nextMove()** Determines the next move of the drone by finding the best move direction among the 16 possible directions that are reachable from the drone's current position. The most desirable move is the move that yields the maximum number of coins. If there are more than 1 moves that are equally the most desirable then the random number generator is used to pick one of these moves.
public String	**getDroneType()** Returns as a String the name of the type of the drone. Namely, "stateless".

2.9 StatefulDrone

The `StatefulDrone` class extends the `Drone` abstract class. It represents a drone within our game that is stateful. This means that the stateful drone's strategy has no limitations in determining the next move. The stateful drone's goal is to connect with all positive stations, while avoiding, whenever possible, connecting to negative stations. For more information on the stateful drone's strategy refer to Section 3 of this report.

Constructor and Description
`public StatefulDrone(Position position, GameMap gameMap, long randomSeed)` Constructs a StatefulDrone object and invokes the constructor of the Drone superclass.

Fields

Modifier and Type	Field	Description
`private Station`	**`targetStation`**	This stateful drone's target station.

Methods

Modifier and Type	Method and Description
`public void`	**`findPath()`** Finds a path to reach all positive stations in the game map. When all such stations have been reached, the drone moves back and forth to its last visited position.
`private LinkedList<Move>`	**`nextBatchOfMovesToTarget()`** Returns a sequence of moves in order to reach the drone's current target station starting from the drone's current position, while avoiding negative stations. The A* path-finding algorithm is used to determine this sequence of moves. The moves are not executed within this method. If the target is unreachable, null is returned. For details refer to Section 3.2.
`private double`	**`calculatePenalty(Node node)`** Calculates the penalty associated with this node. The penalty is positive for nodes with position within the range of a negative station, and zero for the rest of the nodes. For details refer to Section 3.2.
`private LinkedList<Move>`	**`reconstructPath(Node current)`** Constructs and returns the sequence of moves to be followed followed to reach the current node, following the shortest path from the start node. To do this, the nodes' cameFromNode and cameFromDirection attributes are used.
`public String`	**`getDroneType()`** Returns as a String the name of the type of the drone. Namely, "stateful".

2.9.1 StatefulDrone.Node

The Node class is a private static nested class within the StatefulDrone class. It is used as a helper class for the parts of the stateful drone's strategy that use A* search. More specifically, when finding a path from the current position to the position of the target station using the nextBatchOfMovesToTarget() method of the StatefulDrone class. In A* search, each Node is associated to a position. For each position there is at most one Node object associated to it. To ensure this, the constructor is private and Node objects are obtained through the getNodeWithPosition() factory method.

Constructor and Description
private Node(Position p) Allocates a Node object and initialises it to represent the node with the given position.

Fields

Modifier and Type	Field	Description
public static Set<Node>	**allNodes**	All nodes that exist in the current A* search. Initially empty.
public final Position	**position**	The position of this node.
public double	**g**	Distance travelled on the shortest path from the starting node to reach this node. Initialised to positive infinity.
public double	**h**	Estimated distance from this node to the target node. Initialised to positive infinity.
public double	**f**	Estimated distance to reach the target node from the starting node through the shortest path to this node. i.e. f = g + h. Initialised to positive infinity.
public Node	**cameFromNode**	The preceding node on the shortest path from the starting node to this node.
public Direction	**cameFromDirection**	The direction of travel from the previous node on the shortest path from the starting node in order to reach this node.

Methods

Modifier and Type	Method and Description
public static void	**clearNodeSet()** Empties the allNodes set. Has to be called before each new A* search.
public static Node	**getNodeWithPosition(Position pos)** Returns the Node object associated with the given position. If such Node object already exists in the allNodes set, that object is returned. Otherwise, a new Node object is returned and added to the allNodes set.

`public int`	`compareTo(Node otherNode)` Compares this node to another node, based on the f scores of the nodes.
`public void`	`calculateHandF(Station targetStation)` Calculates the h and f scores of this node, given the target station. The h score is the euclidean distance from the position associated to this node to the position of the target station. The f score is calculated as the sum of this node's g score and h score.
`public Map<Direction, Node>`	`expandNode()` Returns a `Map` containing as values all nodes reachable from this node. The keys of the map are the directions followed to reach the respective nodes from the current node.

2.10 GameRules

The `GameRules` class is a class only containing public static final fields that represent constants that dictate the rules of the game.

Fields

Modifier and Type	Field	Description
`public static final double`	`CONNECT_DISTANCE`	The maximum distance allowed between a drone and a station for a connection to be possible.
`public static final double`	`POWER_CONSUMPTION`	The amount of power consumed for one move of the drone.
`public static final double`	`TRAVEL_DISTANCE`	The distance a drone travels with one move.
`public static final double`	`NUM_OF_MOVES`	The maximum number of moves a drone is allowed to make.
`public static final double`	`INITIAL_POWER`	The amount of power a drone has initially.
`public static final double`	`MIN_NEGATIVE_STATION_PENALTY`	The minimum penalty coefficient for a negative station in the stateful drone's strategy
`public static final double`	`TOP`	The top play area boundary.
`public static final double`	`BOTTOM`	The bottom play area boundary.
`public static final double`	`RIGHT`	The right play area boundary.
`public static final double`	`LEFT`	The left play area boundary.

2.11 App

The `App` class contains the application's entry point (the main method).

Methods

Modifier and Type	Method and Description
`public static void`	**`main(String[] args)`** Reads and parses (where necessary) the arguments from the terminal and invokes the `runSimulation()` method.
`public static String`	**`runSimulation(String day, String month, String year, double latitude, double longitude, long randomSeed, String droneType)`** Runs the simulation. Constructs the game map object for the given date, the drone of the given type with the given position. Then, invokes the drone's `findPath()` method to generate the drone's path which is subsequently plotted on the game map. Finally, the drone's move history and the map with the path are saved to files. A string containing the key characteristics of the simulation, including the score of the drone and the perfect score, is returned.

3 Stateful drone strategy

The stateful drone's goal is to connect with every positive station on the map and avoid when possible connecting to negative stations. In contrast to the Stateless drone, it computes batches of moves to be executed to reach each positive station. Each of the moves in a batch is determined having knowledge of the moves that are preceding it in the batch. This allows the drone to move in a more sensible and intelligent manner, preventing unnecessary loss of power by performing random moves.

3.1 High level description of drone strategy

1. The drone queries the game map to find the closest positive station to its position. The distance metric used is Euclidean distance. This station is called '**target station**'.
2. Then, the A* path-finding algorithm (for details refer to Section 3.2) is used to the shortest path (sequence of moves) that leads the drone from its current position to a position that is within range of the target station. The A* algorithm is modified to avoid when possible paths that pass through positions that are within range of a negative station. This path is as short as possible, while avoiding connecting to negative stations.
3. After the sequence of moves is determined, each of the moves is executed. If at any time the maximum number of moves allowed is exceeded or the drone runs out of power, the drone stops.
4. After all of these moves have been executed, steps 1 to 3 are repeated until the drone has connected to all positive stations on the game map.
5. After the drone has connected with all the positive stations, the drone simply moves back and forth to its last visited position, which is a safe move to make. It is not possible for the previous position to be within range of a negative station as it has been visited again and if such station existed it has been neutralised. This continues until it reaches the maximum number of moves allowed or it runs out of power.

Remark: In our game there are no unreachable stations since the map has no obstacles. In the worst case, the drone will just have to connect to a negative station. However, in the implementation of the drone's strategy provision has been made for the existence of unreachable stations.

3.2 Finding a path from a position to the target station using A*

To find a path from the current position of the drone to the position of the target station, a modified version of the A* path-finding algorithm is used. A* is an informed path search algorithm which finds the shortest path between positions on the map.[3] In our case, it is used to find the shortest path from the current position of the drone to the position of the target station. The modification implemented to the algorithm is an added penalty which penalises connections to negative stations along the path. Therefore, when possible the path is not within range of negative stations. If there is no such path, the algorithm ensures that the loss of coins and power is minimised.

For the algorithm we are representing positions on the game map as **nodes** (instances of the `Node` class). Each node is associated with one position, and 3 scores: f, g and h.

- **g** is the distance travelled from the starting position to the node's position on the shortest path from the starting position to the node's position. The negative station penalty is also added to the value of g.
- **h** is the estimated distance from the node's position to the position of the target station. It is calculated using a heuristic function. In our case, this heuristic is the Euclidean distance from the node's position to the position of the target station.
- **f** is the sum of g and h which represents the estimated cost of the shortest path from the starting position to the target station's position through the node.

Each node also keeps a pointer to the previous node on the shortest path to itself (`cameFromNode`) and stores the direction of travel from that previous node to itself (`cameFromDirection`). These are used to construct the sequence of moves that are to be executed by the drone, when the A* search algorithm reaches the target station.

A* uses a priority queue, called **'frontier'** to store the nodes that may be expanded. Each node in the frontier is associated with a path from the starting position (through the `cameFromNode` pointers of the nodes). At each iteration of the main loop of the algorithm, one of these paths is extended, by expanding the relevant node in the frontier. The algorithm favours nodes that are both close to the starting node and close to the target station. Hence, the node with the lowest f score is expanded. Since the g score has the negative station penalty added to it, paths that pass through the range of negative stations are avoided. When all paths pass through negative stations, the ones with lowest power/coin loss are selected (i.e. those with the lowest penalty).

Initially, the frontier contains only the starting node which has g score 0, h score the Euclidean distance to the target station and f score (as always) the sum of g and h. Since it the first node, `cameFromNode` and `cameFromDirection` are null. The default g, h and f scores for all new nodes is positive infinity.

At each iteration, the node with the lowest f score is chosen and removed from the frontier. This is called the **'current'** node. If the current node is within the range of the target station, the main loop is terminated and the algorithm returns the reconstructed path from the start node to the current node. Otherwise, the current node is expanded by finding all the nodes that are reachable from it, taking into account the 16 allowed move directions and the play area boundaries.

For each neighbour, a tentative g score is calculated by adding the move distance from the current node to the neighbour to the g score of the current node. If there is a negative station within range of the neighbour, the relevant penalty is also added to the tentative g score. If the tentative g score is less than the g score of the neighbour it means that the path to the neighbour through the current node is shorter than any other path to the neighbour encountered before. Hence the f and g scores are updated accordingly, `cameFromNode` is set to point to the current node and the associated move direction is stored in `cameFromDirection`. If the neighbour is not in the frontier already, it is added to it.

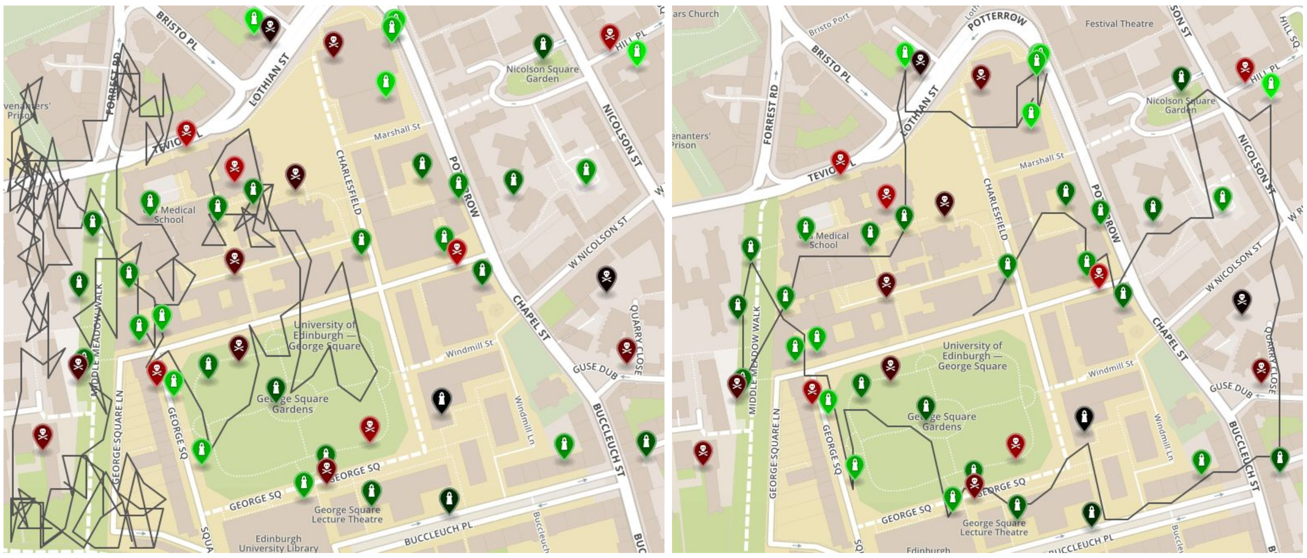
The loop is repeated until there are no nodes left in the frontier or the target station is reached. If there are no nodes left in the frontier it means that the node is unreachable. This should not happen in our case since there are no obstacles in the game map.

Negative Station Penalty

If a node is not within range of a negative station, the penalty associated with it is 0. For negative stations (which have negative coins and power) we calculate *-min(coins, power)* where coins and power are the coins and power of the station that is within range. If *-min(coins, power)* is greater than the minimum negative station penalty defined in the GameRules class, then *-min(coins, power)* is the penalty. Otherwise, the minimum negative station penalty is used.

3.3 Comparison of the paths of the Stateless and Stateful drones

The Stateful drone performs significantly better than the Stateless drone. It achieves perfect score on all 730 test maps provided. It collects all the coins from the positive stations without losing any by connecting to negative stations.



Figures 3.3.1, 3.3.2 The paths followed by the Stateless (left) and Stateful (right) drones for the map of 12/08/2019

References

1. Stevens, P. (2010). *Introduction to Object Orientation*. [online] Available at: http://homepages.inf.ed.ac.uk/perdita/OO/ooBasics.pdf [Accessed 16 Nov. 2019].

2. Perez, S. (2018). *Writing Flexible Code with the Single Responsibility Principle*. [online] Medium. Available at: https://medium.com/@severinperez/writing-flexible-code-with-the-single-responsibility-principle-b71c4f3f883f [Accessed 16 Nov. 2019].

3. Patel, A. (2012). *Introduction to A**. [online] Stanford Theory Group. Available at: http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html [Accessed 1 Dec. 2019].