ILP Report

Daragh Meehan (s1714840)

Contents

1 Software Architecture Description

The Software Architecture of my application broadly follows the Model-View-Controller design pattern. The Controller (PowerGrabController) is accessed at the start of the program from the App launcher class, while the View is the .txt and .geojson files output from the program after the game has been simulated. The Model itself (PowerGrabGame) handles the vast majority of the data and logic of the simulation framework.

The program, once launched in the App class, begins in the Controller. Here the input from the user is validated, ensuring the simulation is run only with valid arguments. The Controller then interacts with the Model through the PowerGrab interface which specifies the three central subsequent procedures of a simulation of a game of PowerGrab - setting up the game, playing it, and finally reporting on the outcome.

PowerGrabGame implements the PowerGrab interface in line with the specification of the coursework documentation, but its actual implementation is hidden from the Controller through the use of this interface, following a façade design pattern. The reason for using both of these design patterns was to encapsulate the simulation framework, completely hiding the data and logic from the user in a cohesive unit, while providing a natural interaction with the system for the user through the use of the Controller's input validation and decoupled interaction with the Model, and the View's output files. The modularity of the system also allows for easier maintainence and evolution in the future.

The Controller calls each of the interfaces methods in turn, after initialising the game. In the setup phase, PowerGrabGame instantiates the objects necessary for the simulation. First, builders for both the Drone and Map objects (DroneBuilder and MapBuilder respectively) are called. These builders (following the Builder creational design pattern) separate the construction of these objects from their representation, encapsulating the non-trivial tasks of creating them, and increasing modularity. DroneBuilder parses the user arguments and constructs the correct drone version, while MapBuilder reads the GeoJSON map files from the web and constructs the feature collection of charging points. If these were successful the List of ChargingStation objects (the charging stations of the map) and MovementLog object are both also instantiated. All of these objects are stored in PowerGrabGame.

The Drone object represents the PowerGrab drone which navigates around the map and charges from charging stations. It is implemented as an abstract class which contains all the basic fields and functionality required by both drone types. The actual Drone instantiated in PowerGrabGame is a subclass of Drone, either StatelessDrone or StatefulDrone, however polymorphism is used to encapsulate the particular drone's decision making - it is only referred to as a Drone object. StatelessDrone and StatefulDrone are implementations of the Stateless and Stateful drone types respectively as described in the coursework document.

The Map object encapsulates the GeoJSON features of the particular simulation's map - it stores the charging station features and a record of the drone's flight path after every move - decoupling these data structures from PowerGrabGame's simulation framework in a cohesive unit. It also provides functionality to reproduce the GeoJSON map with the drone's path added.

The List of ChargingStation objects is constructed by the Map object from its charging station features. Each ChargingStation represents one of the fifty charging stations of our

simulation. They encapsulate all important attributes of a particular station, their position, and coins and power value. The List of ChargingStation objects is stored in PowerGrabGame as it is required by the Drone object to decide its next move(s) and to charge.

Both the Drone and ChargingStation objects' positions on the map are represented by a Position object. Each Position object encapsulates a specific immutable latitude and longitude. The Position class also implements methods related to positions - calculating positions given a direction, if a given position is valid, and the distance between two positions.

In the play phase, the simulation of the game is carried out. Until 250 moves have been made, or the Drone object runs out of power, the drone decides on a move, makes the move, and finally the drone's flight path is added to the Map object and its move is recorded in the MovementLog object.

Each of the drone's moves is represented by a Move object which captures all the details of a move. The MovementLog object records all of the Move objects returned by the drone, allowing them to be written out to a .txt file. Decoupling this functionality from the PowerGrabGame's simulation framework results in increased modularity allowing more flexible altering of these classes and easier maintainence in the future.

In making a move, the Drone object must decide in which direction to move, move in this direction (consuming power), and finally charge from the closest ChargingStation object if it is close enough to do so. The direction of a move is captured by one of the 16 constants in the Direction enumerated class. This is a natural representation of the 16 possible directions of movement.

The decision of move direction of the drone depends on the type of the Drone object. The stateless drone makes a pseudorandom decision of what direction to move in with no regard to its past moves, while the stateful drone analyses the map to produce the best strategy it can. The full details of the stateful drone's strategy are explained in section 3 of this document.

In calculating this strategy, the StatefulDrone object runs basic simulations of PowerGrab, represented as PowerGrabSimulation objects. The PowerGrabSimulation class is an abstract class that extends the PowerGrab interface (it is a subclass). It was natural that these basic simulations follow the general procedure of a simulation of a game of PowerGrab, and they also provide the necessary functionality for the StatefulDrone object to make its decision about which strategy to use. Each basic simulation is an instance of a subclass of PowerGrabSimulation but is referred to as a PowerGrabSimulation object. The use of polymorphism encapsulates the internal strategy-forming algorithm of the particular basic simulation. The three basic simulation algorithms I have implemented are encapsulated in the NearestNeighbourSimulation, NearestInsertionSimulation, and FarthestInsertionSimulation classes.

Each basic simulation also uses the StatefulDrone class's A* search algorithm to calculate its move schedule. My implementation of the algorithm uses Node objects and a NodeComparator to order these nodes faithful to the algorithm.

Finally, in the report phase, the output files are written - the Map object produces the GeoJSON map with both the charging station features and the flight path of the drone, and the

MovementLog writes the record of each of the Drone's moves. These constitute the View of the MVC pattern, allowing the user to see the results of the simulation.

The GameParameters class consists of many of the constant values of the PowerGrab game (e.g. the distance travelled in each move or the drone's starting power) allowing these to be altered for testing (e.g. for testing a low power or high power-consuming drone) or to change these constants without having to alter the code of the simulation.

2 Class Documentation

The Class Documentation provides concise documentation for each class in the application, starting with the entry point and continuing generally in the order they are called/used. Trivial fields and methods are omitted, as are basic unit test classes. Only one implementation of PowerGrabSimulation is included - the others follow the same structure.

2.1 App

`public class App`
This is the entry point of the program.

Method Summary

`public static void main(String[] args)`
: Immediately calls PowerGrabController's playPowerGrab method with user supplied arguments.

2.2 PowerGrabController

`public class PowerGrabController`
Manages the high level control flow of the program.

Method Summary

`public static void playPowerGrab(String[] args)`
: Validates the user input, instantiates the game, then performs setup, plays the game, and reports the results.

2.3 InputValidator

`public class InputValidator`
Validates the user's input.

Method Summary

`protected static boolean isValid(String[] args)`
: Verifies that the user's input is in a valid format and that the starting position of the drone is in the play area.

2.4 PowerGrab

`public interface PowerGrab`
A simple interface that specifies the three subsequent procedures of a game of PowerGrab.

Method Summary

`public void setup()`
Sets up the simulation.

`public void play()`
Plays the game.

`public void report()`
Reports on the outcome of the game.

2.5 PowerGrabGame

`public class PowerGrabGame implements PowerGrab`
This is our standard high-level implementation of a game of PowerGrab as specified in the coursework document. It performs setup, plays the game, and reports the outcome.

Field Summary

`private Drone drone`
Our stateless/stateful drone

`private Map map`
The PowerGrab map as represented by a list of features.

`private List<ChargingStations> chargingStations`
The charging stations of the game.

`private MovementLog`
A log which records the moves of the drone.

Method Summary

`@Override public void setup()`
Sets up the simulation by calling the Drone and Map builders, and instantiates the chargingStations and movementLog. The game is terminated if the Map build is unsuccessful.

`@Override public void play()`
Keeps getting the drone's next move and records it until the drone runs out of power, or the maximum moves have been made.

`@Override public void report()`
Reports on the outcome of the game by writing the .txt and .geojson output files.

2.6 MapBuilder

`public class MapBuilder`
This class builds the PowerGrab map for the specified date by reading the features from the web and creating a FeatureCollection used in the Map constructor.

Method Summary

`public static Map build(String day, String month, String year)`
Formats the input string creating a valid URL, connects and reads the map's features, creates a feature collection, and constructs the map.

`private static HttpURLConnection convertToConnection(String mapString`
Converts the URL String to HttpURLConnection and opens the connection.

`private static void connect(HttpURLConnection conn)`
Connects to the URL.

`private static String readMap(HttpURLConnection conn)`
Reads the features from the address to use in creating the map's FeatureCollection.

2.7 Map

`public class Map`
This class stores the features of the GeoJSON representation of our PowerGrab map - the charging stations and the drone's flight path.

Field Summary

`private List<Feature> mapFeatures`
The list of charging station features, to which the drone's path is added.

`private LineString dronePath`
A record of the drone's flight path.

Method Summary

`public List<ChargingStation> getChargingStations()`
Builds a list of charging stations from the features to allow processing by the drone to choose it's moves.

`public void addDronePath(Position positionBefore, Position positionAfter)`
Adds a move to the Drone's flight path.

`public void createGeoJSONMap(String day, String month, String year, String droneVersion)`
Builds a GeoJSON representation of the PowerGrab map with the drone's flight path included with the charging stations.

2.8 ChargingStation

`public class ChargingStation`
This class represents the charging stations of PowerGrab. Each station has an immutable position, and coins and power values to charge the drone.

Method Summary

`public boolean isInRange(Position position)`
Determines if the drone is close enough to charge.

`public void transferCoins(float coinsTransfer)`
Makes a transfer of coins when the drone is charged.

`public void transferPower(float powerTransfer)`
Makes a transfer of power when the drone is charged.

2.9 MovementLog

`public class MovementLog`
The movement log records each of the drone's moves, and writes the log of moves to a .txt file.

Field Summary

`private List<Move> moves`
The moves of the drone.

Method Summary

`public void addMove(Move move)`
Adds a move of the drone to the log.

`public void writeLog(String day, String month, String year, String droneVersion)`
Writes the log of moves to a .txt file.

2.10 Move

`public class Move`
Represents a single move of the drone. Each move can be formatted to be printed in a log of moves.

Field Summary

`public final Position positionBefore, positionAfter`
The position of the drone before and after the move.

`public final Direction moveDirection`
The direction of the move.

`public final float coinsAfter, powerAfter`
The drone's coins/power after the move.

Method Summary

`@Override public String toString()`
Formats a move to be used in a log of moves.

2.11 DroneBuilder

`public class DroneBuilder`
Builds a stateless or stateful drone with the given starting position and seed.

Method Summary

`public static Drone build(String initLatitudeAsString, String initLongitudeAsString, String seedAsString, String droneVersion)`
Parses the arguments and constructs the right type of drone. Assumes format has been verified by InputValidator.

2.12 Drone

`public abstract class Drone`
The Drone of the PowerGrab simulation, instantiated by a StatelessDrone or StatefulDrone object.

Field Summary

`private Position position`
The position of the drone.

`private float coins`
The drone's coins.

`private float power`
The drone's power

`private final Random rnd`
Used by the drone to make a pseudorandom choice of direction decision.

Method Summary

`public Drone(Position position, int seed)`
The drone's constructor. Instantiates the initial position, coins and power value, and rnd instance.

`public boolean canMove()`
Calculates if the drone has enough power to move.

`public Move makeMove(List<ChargingStation> chargingStations)`
Determines the drone's next direction to move in, moves in this direction, charges from the nearest power station if in range and constructs a Move object with the details of the moves.

`public abstract Direction chooseDirection(List<ChargingStation> chargingStations)`
The direction chosen is determined by the individual drone's strategy algorithm.

`private void consumePower()`
Consumes power after a move.

`private void charge(ChargingStation chargingStation)`
Charges the drone from a given station. Assumes this station is the closest and in range.

`public Direction chooseRandomDirection(List<ChargingStation> chargingStations)`
Makes a pseudorandom decision of the direction to move in favouring nearby positive stations and avoiding negative ones.

`public static List<Direction> calculateAvailableDirections(Position position)`
Calculates the available directions to move in for a given position.

`public static ChargingStation calculateClosestStation(Position position, List <ChargingStation> chargingStations)`
Calculates the closest charging station to a given position.

`private List<ChargingStation> calculateNearbyStations(List<ChargingStation> chargingStations)`
Calculates the stations possible to visit in only one move to avoid calculating for every station when making a pseudorandom decision about the direction to move in.

`private Direction makeRandomChoice(List<Direction> directions)`
Chooses pseudorandomly between the given directions using the Drone's Random rnd instance.

2.13 StatelessDrone

`public class StatelessDrone`
The implementation of the stateless drone as specified in the coursework document.

Method Summary

`@Override public Direction chooseDirection(List<ChargingStation> chargingStations)`
Uses the abstract Drone class's pseudorandom decision of move direction to make its choice.

2.14 StatefulDrone

`public class StatefulDrone`
The implementation of the stateful drone specified in the coursework document.

Field Summary

`private List<Direction> nextMoves`
The move schedule of the drone - the strategy of the best basic simulation.

Method Summary

`@Override public Direction chooseDirection(List<ChargingStation> chargingStations)`
Forms the stateful drone's strategy if not yet chosen. Returns the next move direction in the move schedule, or a pseudorandomly chosen direction if the move schedule is empty.

`private void formBestStrategy(List<ChargingStation> chargingStations)`
Runs three basic simulations of the PowerGrab map and chooses the best to determine its best strategy.

`private static List<Direction> chooseBestStrategy(List<Float> simulationResults, List<List<Direction>> simulationMoves)`
Chooses the best strategy from all the basic simulations.

`public static List<Direction> findShortestPath(Position startPosition, ChargingStation goalStation, List<ChargingStation> chargingStations)`
Finds the optimal path to a charging station using A* search.

`public static double aStarHeuristic(Position currentPosition, Position goalPosition)`
The admissible and consistent heuristic used by A* Search.

`public static List<ChargingStation> calculatePositiveStations(List<ChargingStation> chargingStations)`
Calculates the positive stations from a given List of ChargingStation objects.

`public static double[][] calculateDistanceMatrix(List<Position> positions)`
Calculates the distance matrix of a given List of Position objects.

`public static double calculateTotalRouteDistance(int[] stationOrder, double[][] distanceMatrix)`
Calculates the estimated total route distance of a route order.

`private static int[] reverseRouteSegment(int[] order, int i, int j)`
Reverses the segment of the route order from index i to j.

`private static int[] swapRouteSegments(int[] order, int i, int j, int k)`
Swaps the segments of the order from index i to j - 1, and j to k - 1.

`public static int[] twoOptOptimise(int[] stationOrder, double[][] distanceMatrix)`
Optimises a route order using the 2-opt algorithm.

`public static int[] threeOptOptimise(int[] stationOrder, double[][] distanceMatrix)`
Optimises a route order using the 3-opt algorithm.

2.15 PowerGrabSimulation

`public abstract class PowerGrabSimulation implements PowerGrab`
The basic simulation ran by the stateful drone.

Field Summary

`private List<int[]> stationOrders`
The different route orders.

`private float bestResult`
The best simulation result.

`private List<Direction> bestMoves`
The best simulation move schedule.

Method Summary

`@Override public void setup()`
The setup phase of the basic simulation.

`public abstract List<int[]> chooseStationOrders()`
Calculates the different route orders using the simulations algorithm.

`@Override public void play()`
The play phase of the basic simulation.

`private static int findStationIndex(ChargingStation chargingStation, List<Charging Station> chargingStations)`
Finds the index of a given charging station.

`public static int findBestInsertionPosition(int[] stationOrder, double[][] distanceMatrix, int stationToInsert, int numberOfStationsVisited)`
Finds the position to place a given station in the route order that minimises the increase in estimated distance.

`public static int[] insertStationIntoRouteOrder(int stationToInsert, int best InsertionPosition, int[] stationOrder)`
Inserts a given station in the route order in the specified position.

2.16 NearestNeighbourSimulation

`public class NearestNeighbourSimulation extends PowerGrabSimulation`
A basic simulation that uses the nearest neighbour algorithm to decide its route order.

Field Summary

`private int[] basicOrder`
The basic route order.

`private int[] twoOptOptimisedOrder`
The basic route order optimised with 2-opt.

`private int[] threeOptOptimisedOrder`
The basic route order optimised with 3-opt.

Method Summary

`@Override public List<int[]> chooseStationOrders()`
Chooses the route order using the nearest neighbour algorithm.

`@Override public void report(`
The report phase of the basic simulation.

`private static int[] calculateNearestNeighbourOrder(double[][] distanceMatrix)`
Calculates the nearest neighbour route order.

2.17 Node

`public class Node`
Represents a node in our A* search algorithm.

Field Summary

`public final Position position`
The immutable position of a node.

`private List<Direction> path`
The series of moves to reach the node.

`private double fScore`
The f-score of the node (g-score + heuristic).

`private int gScore`
The g-score of the node.

Method Summary

`public Node(Position position, List<Direction> path, Position goalPosition)`
The Node constructor. Sets the g-score equal to the length of the path, and calculates the f-score.

`public boolean reachedGoal(ChargingStation goalStation, List<ChargingStation> chargingStations)`
Calculates if the given node has reached the goal.

`public List<Node> getNeighbours(Position goalPosition)`
Calculates the neighbouring nodes of a given node.

2.18 NodeComparator

`public class NodeComparator implements Comparator<Node>`

Method Summary

`@Override public int compare(Node n1, Node n2)`
Compares two nodes - nodes are ordered based on their f-score.

2.19 GameParameters

`public class GameParameters`
This class consists of all the parameters of PowerGrab. Changing these values allows us to tweak how the game works (e.g. by making the move distance smaller or increasing the power cost of a single move).

2.20 Position

`public class Position`
Represents a position in our game. The constants defined specify the move distance and changes to latitude and longitude by different angled moves, and the bounds of the game's map.

Field Summary

`public final double latitude, longitude`
The immutable latitude and longitude values the specify the location of a position.

Method Summary

`public Position nextPosition(Direction direction)`
Gives the resulting position after a move in a certain direction.

`public boolean inPlayArea()`
Determines if the position is within the play bounds.

`public static double calculateDistance(Position p1, Position p2)`
Calculates the euclidean distance between two positions.

2.21 Direction

`public enum Direction`
This class consists of enum values representing the 16 different directions the drone can move in.

3 Stateful Drone Strategy

My stateful drone (StatefulDrone object) runs basic simulations on the given map using different strategies, finds which one should result in the maximum coins in the shortest amount of moves, and adds these to its move schedule, all before making its first move. These are the moves returned by the drone when queried by PowerGrabGame for its next move direction as long as there are moves left in the schedule. After this, the drone will make a pseudorandom decision on its next direction - using the same strategy as the stateless drone (a StatelessDrone object).

When first queried for a move, the stateful drone forms its best strategy. It does this by setting up three different basic simulations of the game, however all are referred to as PowerGrabSimulation abstract objects, encapsulating their internal functionality and making future maintainence and design of new basic simulation algorithms straightforward. Following the PowerGrab interface, every PowerGrabSimulation object performs setup, playing and reporting, but also can return the result of the simulation (the coins collected) as well as the corresponding move schedule. The setup and play phases of the basic simulations are all the same (excluding the specific strategy making), and are implemented in PowerGrabSimulation.

When instatiating a PowerGrabSimulation object and in the setup phase, its necessary fields are instantiated - most importantly the distance matrix. The distance matrix gives an approximation of the drone's travel distance between any two of the route positions (the starting position of the drone and position of every positive charging station). These are calculated as the euclidean distance between the points, and don't take into account the obstacles that might be in the way (negative stations) or that the drone cannot fly in any arbitrary direction. The distance matrix is used in all basic simulations to determine the order in which to visit the positive stations (the route). At the end of the setup phase, the different move orders chosen by the particular basic simulation's strategy are calculated.

I have implemented three different tour construction algorithms (nearest neighbour, nearest insertion, and farthest insertion) for finding a route, and two tour improvement algorithms (2-opt and 3-opt) for trying to improve the efficiency of a route. The tour construction algorithms are specific to each subclass of the PowerGrabSimulation class, while the tour improvement algorithms are general algorithms that can be used on any route order.

A NearestNeighbourSimulation object calculates its route order based on the nearest neighbour algorithm described here. The route order starts with the drone's initial position and the closest station to it. Then the next station added to the order is the closest station to the latest station in the order, extending the route, repeated until all stations are added.

The NearestInsertionSimulation and FarthestInsertionSimulation objects calculate their route orders based on the nearest insertion algorithm and farthest insertion algorithm described here and here respectively. The route order of the NearestInsertionSimulation object starts with the drone's initial position and the closest station to it, while that of the FarthestInsertionSimulation object starts with the drone's initial position and the farthest station from it. The next station added to the order is the closest station for the NearestInsertionSimulation and farthest station for the FarthestInsertionSimulation to any already in the order or to the drone's initial position. The chosen station is added to both orders in the position that increases the route's overall estimated distance the least.

Once the basic route order has been chosen, all the basic simulations form two more orders - the basic order is optimised using both a 2-opt and 3-opt route optimisation algorithm based off of the algorithms described here and here respectively. My 2-opt algorithm loops through the basic order until no improvements are made looking at the segment of the route between every pair of stations in the order and reversing any segment that will result in a lower estimated distance. My 3-opt algorithm works very similarly but instead looks at ways of rearranging two subsequent segments at once by reversing one segment or the other, reversing the combined segment, or swapping the order of the segments, making the change to the order only if it lowers the total estimated distance. Once these three orders have been calculated, the PowerGrabSimulation play phase begins.

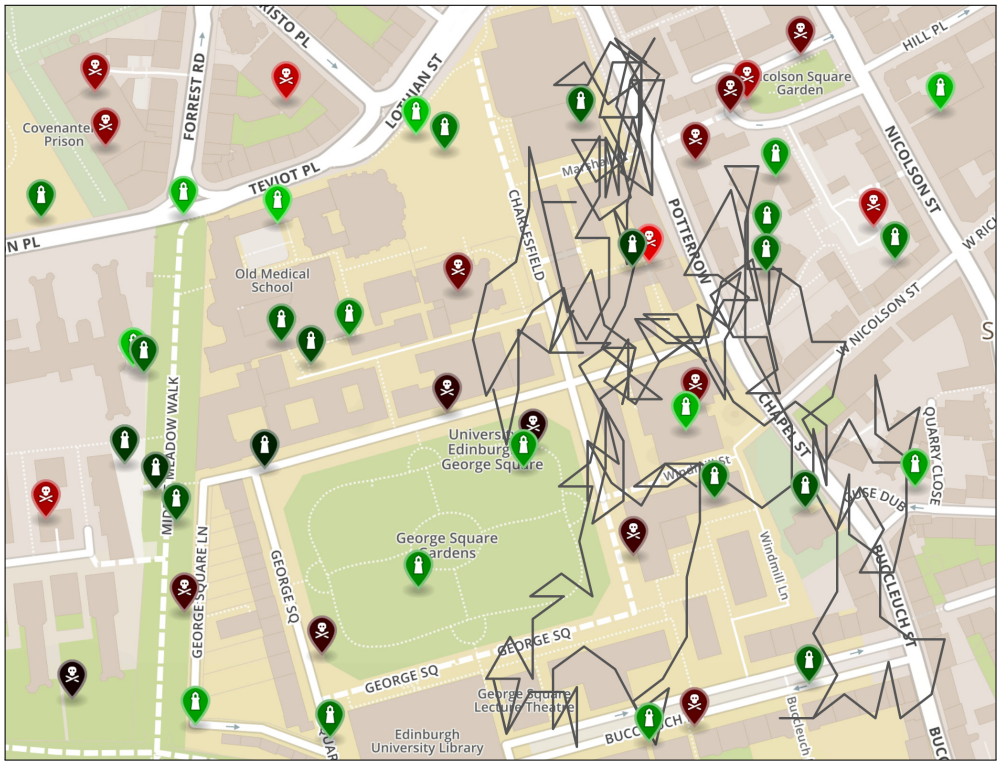
In the play phase, the basic simulations of the game are carried out. It should be noted that all basic simulations treat positive stations equally, so don't favour visiting any over any others, and also charging is not modelled. The basic simulation tries each route order in turn and finds the best one, assigning that to its bestMoves field, and its estimated result to its bestResult field. In trying a route order the next station in the route order not already visited is found, and A* search algorithm returns the optimal set of moves (Direction constants) to get to this station. Any time a station has been reached its coins are added to the result. This process loops until the maximum move count has been reached, or all stations have been visited.

My A* search algorithm is based off of the pseudocode here, finding a move schedule/path that takes the drone from any position to within range of and closest to the desired charging station. My heuristic is the euclidean distance between the starting position and the position of the charging station, which is admissible and consistent and so finds an optimal path without processing a node more than once. My g-score for each node is the minimum moves required to reach this node so far. The algorithm never returns a path which would result in charging from a negative station, which would result in the drone collecting a sub optimal amount of coins if a map required going through a negative station to visit a positive station giving a net increase in coins.

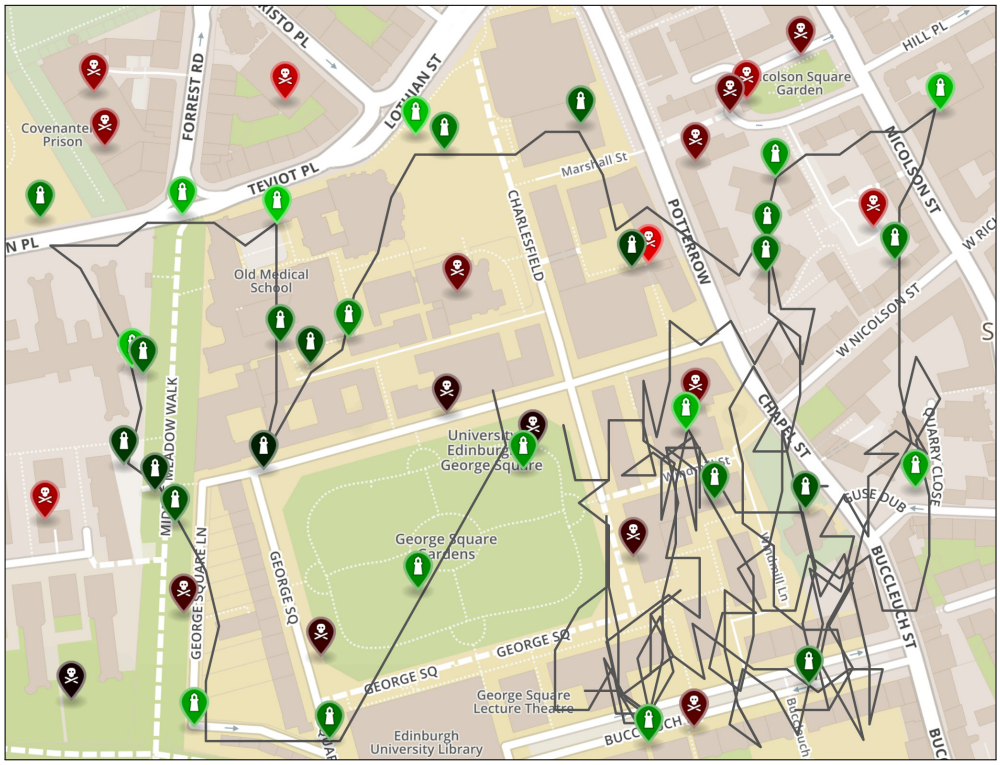
In the report phase, the basic simulation prints the estimated distance of all route orders. The StatefulDrone object chooses its best strategy based on the results of the best route orders of these three basic simulations. It chooses the one with maximum coins, and if multiple are equal, that which is most efficient (least moves). It then adds the moves of the best basic simulation to its move schedule, returning the next move whenever queried by PowerGrabGame.

If this move schedule has been emptied the StatefulDrone then uses the strategy used by the StatelessDrone. The Drone object looks at all its possible move directions and chooses the one that will result in charging from the best (in terms of coins) positive station (if any exist), the least negative (again in terms of coins) station (if all directions will result in charging from a negative station), or else makes a pseudorandom decision of its neutral moves (results in charging nowhere or at a neutral station) using the Drone objects' Random rnd instance.

The flight paths of a stateless and stateful drone on the same map (01/01/2019) are given below.



Stateless Drone (01/01/2019)



Stateful Drone (01/01/2019)