Report on PowerGrab implementation

Informatics Large Practical

December 6, 2019

Table of Contents

Contents

pages

I. Overview

PowerGrab is similaration framework for an autonomous (“smart”) component which is to play against a human component in a location-based strategy game. This component competes against a human player who is acting as the pilot of a remote-controlled (“dumb”) drone which moves only in response to directions from the pilot.

This report describes the design and implementation of the powergrab’s both stateless drones, which is limited in that it is ‘memoryless’ (ie. its decisions doesn't depend on previous moves) and has limited look-ahead (ie. it only knows a state of its current position and the next possible positions), and stateful drone, which has the ability to scan the whole map and learn from previous moves to make sophisticated decisions.

The evaluation for both agents (Stateless and Stateful Drone) is measured in terms of amount powerCoin they collect, but also they have to maintain enough power that will last them for the maximum of 250 moves.

II. Software architecture description

In addition to **Direction, Position, and App,** PowerGrab system has the following additional classes: **Drone**, **StatefulDrone** with its inner class **EvictingQueue** data structure, **StatelessDrone**, **StationsMap** with **Station** and **MapDate** as their static inner classes, **CommandArgsParser**, and IOUtils.
Below is the UML diagram that shows associations, multiplicities, types, classes’ members, and how messages are exchanged between their objects. Please note that getters setters functions were intentionally omitted to have a concise diagram. Also for “dataType($number)” shows that a function takens a number of $number such data type.

UML Diagram for PowerGrab Architecture

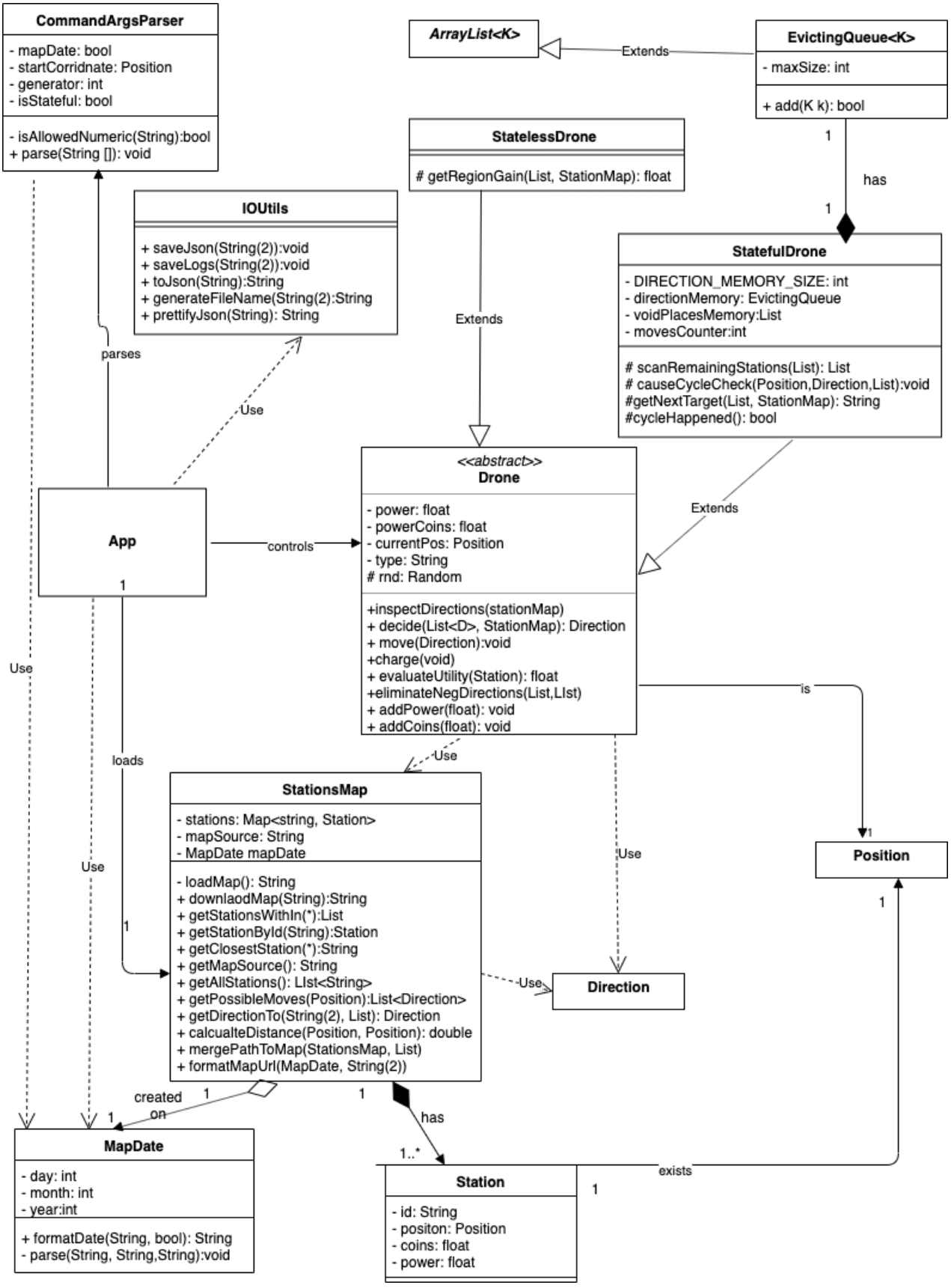

Figure1: UML diagram for PowerGrab classes

The Data flow of the PowerGrab simulation

App receives command line arguments, using **CommandArgsParser** it parses the given input and initializes an instance of **StationMap** on the specified date. It also initializes **Drone** depending on whether it is **StatelessDrone** or **StatefulDrone**, then it call inspect, decide, move, charge functions until the drones makes 250 or runs out of power (whichever comes first). **Drone** takes a **StationMap instance** and uses it while making a decision of where to move. Lastely **App** merges the path to the original map and saves it as a .geoJson file and saves a text file of logs of the moves the drone made.

Class Hierarchy and Rationale behind the design

Drone class is an abstract superclass for both **StatelessDrone** and **StatefulDrone.** Drone class don't only contains common functionalities both drones need, it also enforces, through abstract functions, all functionalities that each class that inherits it should implement. This allowed me to maintain consistency across both drones and avoided redundancy of codes for shared functionalities. And most importantly, this design provided an external interfaces needed to control any drone while also having different internal structures in terms of how they make decisions and what they are capable of. For instance, in the simulation class **App, Drone** could be initialized as either stateless or stateful drone depending on arguments passed - Polymorphism. In the **App** class, I had to focus only on how to control any drone, not a particular drone, which also reduced amount of codes I wrote, but also allow the maintainability in case other types of drone are added in PowerGrab as well as improved abstraction of my codes, which is useful, because anyone if the team can either extend or improve the functionality of either drone or any other function or or just use it without knowing how it is implemented.

StatefulDrone class has the **EvictingQueue** as an inner class and is a private inner class since it is only used inside StatefulDrone. They also one to one relationships since it acts as a memory to keep the recent 4 moves the drone made. I implemented my own EvictingQueue since I only wanted extends a very simple functionality, so I importing a whole package that does the same thing would have increased unnecessary load to my code.

StationsMap represents a map where stations information can be accessed by the **Drone's objects** and it handles all functionality related to locations of either drone and stations as well as all functionalities related to determining locations and directions. Since in the powergrap we care about stations and locations of the drones relative to stations, I found it best to create a specific Map for stations instead of first creating a general map class that may include the other objects other than stations, but it suffices for the purpose of the Powergrab and it can easily be extended.

Station class is a public static inner class of **StationsMap,** with a composition relationship since stations cannot exist outside of the map, and it is static since no internal members of Station need to use **StationsMap,** which also save memory in case many objects of StationMap are initialized(that doesn't happen in this case of powergrab). It is public because it has to be accessed by the drone.

MapDate is also a static inner class **StationsMap ,** with **an** Aggregation type of relationship since MapDate object can be created created by other objects(ie. **CommandArgsParser)**. I created **MapDate** because it is needed in various formats for formatting URL for which to download maps and for saving file names, and it makes sense for any **StationsMap** to have a date of the map it holds.

Additionally, **CommandArgsParser** class handle input arguments for the simulator and parses and validates them, store them in a format that is suitable for efficient computation. It only interacts with **App** class

Lastly, I created **IOUtils**, which is a Utility class with only static Methods that handles functions that are not associated to any objects, but all are related to IO or saving .txt file or geoJson files, with functions for formatting(Prettifying) json into friendly indentation and structure while saving it as well as formating name of files to be saved. It only interacts with **App** class

III. Stateful drone strategy

To improve the score of the my stateful drone relative to stateless drone's, the stateful drone exploits three main facets:

- its ability to scan the whole map and find the best station to visit at each step, which is determined by number of moves it has already made, distance and number of coins and power a station has.
- its ability to determine cycles when they happen by remembering the last four moves it made as well as the places that caused cycles (void places) and try to avoid them.
- It's navigation ability of avoiding negative stations, if possible(same for Stateless), but also be able to dynamically its decision of the target to go to along the way.

What is remembered

- Last four moves (directionsMemory): This is an evicting queue, that keeps only the four recent moves a drone has made. Four last moves is sufficient to determine whether it is trapped by negatives stations since it uses greedy way of determining the best stations to visit. Hence the only cycle that will happen is going back and forth in only two directions.
- **Void places:** When the drone determine a cycle at a particular position, it stores the current position and the position it was before the current position, that is the two places that are causing a cycle. Storing only one of the positions might be enough, however eliminating the possibility to even try to go the previous places helps it escape the negative stations much quicker. I will explain below how this stored void places are used to help us achieve this goal of areas concentrated with only negative stations (especially when they make a curved hole that can't be simply be be escaped by changing a direction by 90 degrees) in navigation section below.
- **Number of moves (time):** The drone also keeps the number of moves it makes as it helps in determination of utility of stations, where the number of moves are direct proportional to the stations' coins and inversely proportional to stations's power and the distance between the current position of a drone and the stations.

Target Determination

To find the next target to visit, the drone determines the remaining the remaining unvisited positive statations and find the best stations by finding one with the minimise cost and maximises utility. The cost of visiting a station is determined by the distance of the station

relative to the drone divided by its utility which is determined by its power, powercoins and the number of moves a drone has made so far.
Let
time= number of moves of the drone
Then
Utility of a station = station's coins * time + station's coins/ time
Cost of visiting a station = distance of a station relative to the drone / utility of station
This const function doesn't only take into account the cost of visiting a station, but also a station that maximizes utility by dividing the utility of a station. That means stations with high utility will have less cost.

Navigation Algorithm

The navigation of the drone uses the cost function as heuristic for determining the best target to visit, however, that does not tell us whether there is no obstacles (negatives stations) if it follows the shortest path to the target or whether the direction of the moves will will get us closer to the target compared to zero stations(which in this case we would charge on stations with zero utility not the target), the same applies to negative stations.

The following is of the algorithm of how the drones navigates employing its memory and cost function to get to a desired target and how desired target dynamically may change.

- The drone first ask from the station Map, the possible legal directions it can move from its current position on the map.
- After it determines the best target (using the method described above) and makes local decision in finding the best direction:

  The best direction is the one that gets it closer to the target, that doesn't not lead to the negative stations vicinity(0.00025 degree), and if the target and a zero value stations are in charging zone(0.00025) that direction leads it closer to positive stations than to the zero-value station, and finally that directions won't lead to the void places the drone have been before and found cycles.

- The drone only makes local decision by scanning to check whether they are stations within 0.00055 degree. If there are no stations is chooses the direction that get it closer to the target. Otherwise, it remove directions that doesn't not meet the rules mentioned above and then chooses the best directions that can it closer in the remaining choice.

  If there is no other choice(which is rarely or if it ever happens -- It can only happen if a drone is initialized around the negative stations, because otherwise it would choose a path that lead it there, or block the possibility of moving back there), it will choose the direction that gets it closer in the original possible directions (even if that would mean charging to a negative station).
- while avoiding the void places(position that leads to cycles), it avoid being in 0.0001 degree of those position. Why 0.0001?

  0.0001 is an optimal number (one of the optimal numbers for the powergrab) because the drone's move has a radius of 0.0003, so the drone has options of jumping over those void positions, in other words they don't entirely block it from moving to other positions but also they don't allow it to be in the positions or very close position to the void position that will lead cycles.

- Note that the after the drone scan the stations within 0.00055 degree all of its rest decisions are made from only stations found from that range, so it only scan the whole map once per move, which is computationally efficient but also effective to allow the drone to make good decisions while avoiding negative stitions.

- When the drone finishes all positive stations it makes random moves around the map while also avoiding only being in the charging range of negative stations.

Below are the two images that shows navigation of stateless drone and how the the stateful drone using the above the strategy was able to visit all stations. (In fact, by testing on all maps of generated for 2019, and 2020, this strategy was was able to achieve a perfect score for all maps with any random generator seed)

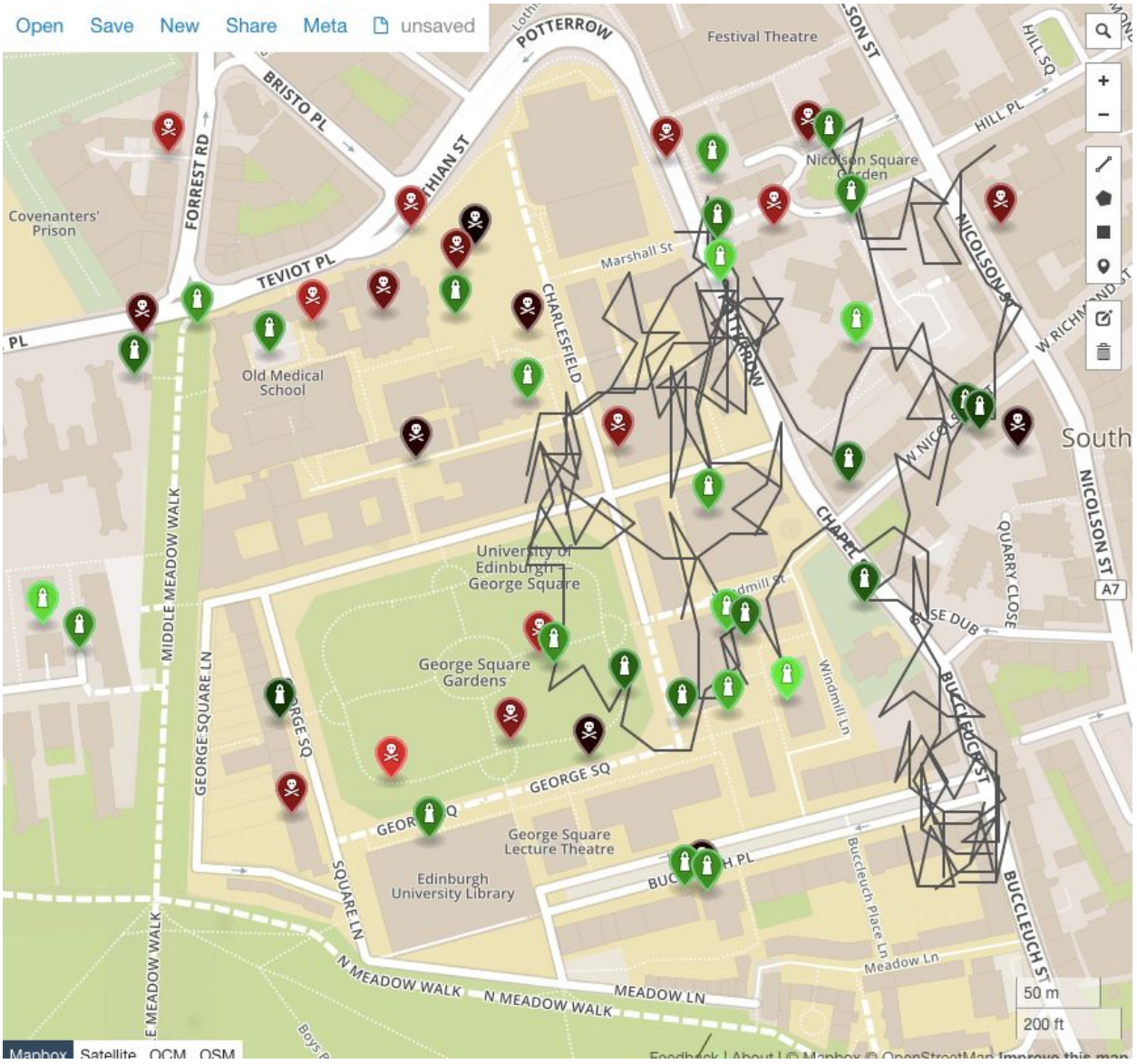

Figure 2: Stateless drone navigation path for 05/05/2019, on5678 random seed

Figure3: Stateful Drone Navigation Path on 05/05/2019 map, 5678 random seed

IV. Class documentation

Note: Please note that this documentation doesn't include getters and setters functions but the classes have them where necessary.
Both data and functions members of class are written in this format:

All of the following classes are in uk.ac.ed.inf.powergrab package.

Modifier and Type : Description

- **abstract class Drone :** extends **java.lang.Object**

a Drone class is an abstract class that represents a generic type of drone that has power, coins, and in other additional to abstract functionalities any drone that extends it should implement, it contains also other common functionalities that are common to all drones.

Fields:

private float power : *power the drone has*
private float powerCoin : *coins the drones has collected so far*
String type : *name of type of a drone*
Position currentPos : current position of a drone
protected Random rnd : random generator seed object

Methods:

public Drone(**Position** initialPostion, int seed, String type): *constructor for the drones that takes its initial position on the map, random generator seed, and the type of the drone*

abstract public ***Direction*** Decide(List<**Direction**> possibleDirection, **StationsMap** stationsMap) : an abstract *function that each drone should implement which return Direction a drones moves to.*

abstract protected float evaluateUtility(**Station** station) : function to be implemented by each drone to determine the utility of station depends on how a particular drone evaluate it.

protected List<**Direction**> inspectDirections(**StationsMap** stationsMap) : *this function takes stationMap and return legal directions a drone can move among 16 possible directions in the stationsMap taking account the drone's current position.*

static protected void eliminateNegDirections(List<**Direction**> negativeDirections, List<**Direction**> directions): *Eliminates negative directions (*negativeDirections*) in given directions. They are all passed by reference.*

private void addPowerCoins(float coins): *adds coins to drone's current coins and makes sure that drones doesn't have negative coins when it connects to negative stations with power that outweigh its current power.*

public void addPower(float p): *add power to the current drones and makes sure it doesn't get negative power when connecting to negative stations in the same way as* addPowerCoins().

public Position move(*Direction* to): given a direction, *it moves a drone to a given direction by 0.003 degree*

public void charge(StationsMap stationsMap): *handles the charging of a drone, when it to any nearest type of station within 0.00025. Also, handles the cases when the negatives power or coins of station outweigh the drone's current power or coins*

- **class StatelessDrone**: extends **Drone**

  represents a stateless drone that is limited in number look aheads of moves it can make and can't remember the number or kind of moves it previously made.

  Methods:

  public StatelessDrone(**Position** startPos, String type, int seed): *a Constructor the Stateless Drone only calls the constructor of its superclass **Drone**.*

  @Override public *Direction* Decide(List<**Direction**> legalDirections, **StationsMap** stationsMap): *given the legal moves, it decide the best directions to move to by only deciding the best moves that takes to an area of maximum utility, not station within its next possible moves, it chooses randomly among the legal directions.*

  protected float evaluateUtility(**Station** station): *evaluates a utility of a stations favoring number of coins and penalizing its power by a factor of 20 percent of the maximum coins a station can have. this because it is memoryless: it does not know the state of the game.*

  private float getRegionGain(List<String> stations, **StationsMap** stationsMap): *given stations ids, it return sum of their utilities. This is used for stations within reach of drone when it's evaluating utility moving in that particular direction.*

- **class StatefulDrone**: extends **Drone**

  represents a sophisticated limitless drones that avoid negative stations and try to collect as many coins as it can it a limited time, which is achieved by the strategy mentioned above.

  Inner Classes:

  private EvictingQueue<**Direction**> directionsMemory : holds past moves the drone made.

  Fields:

  final int DIRECTION_MEMORY_SIZE: *number of past recent moves the drone remembers*

  private int movesCounter: store how many moves the drone has made.

  private List<**Position**> voidPlacesMemory: store the positions that led to cycles.

  Methods:

  public StatefulDrone(**Position** startPositon, String droneType, int randomSeedGenerator): *a Constructor that instantiate the superclass as well as initialize the data members of the Stateful class.*

  @Override public *Direction* Decide(List<**Direction**> possibleDirections, **StationsMap** stationsMap): implements the stateful drone strategy described above. It makes a sophisticated decision of the direction to move to considering the cost of the current target and number of remaining moves, and avoiding negative stations. It also can dynamically changes it target if it meets obstacles.

  protected List<String> scanRemainingStations(**StationsMap** stationsMap): return positive unvisited stations.

  private void causeCycleCheck(**Position** position, ***Direction*** dir, List<**Direction**> directionToAvoid) : *determines if a particular directions at particular position will cause a cycle when a drone goes to a position in voidPlacesMemory, if adds to the directions that should be avoided.*

  private String getNextTarget(List<String> positiveStations, **StationsMap** stationsMap): *return the id of the best station to visit of a least cost and that also maximizes utility.*

  protected float evaluateUtility(**Station** station): *return the utility of a stations described above the the drone strategy sections, which takes into account the amount of coins and power stations have and proportionate them to the number of moves it has already made.*

private boolean cycleHappened(): *return true if a cycle have happened, given that directions memory only keeps four directions. It determine when a drone has made four consecutive alternative moves only two types (for example, N, S, N, S is cycle, but NNNN and NSSN are not)*

- **class EvictingQueue<K>: extends** ArrayList<K> and is an inner class of **StatefulDrone** represents a queue data structure that can only hold a limited number of elements, and pop out the oldest

  Fields:

  private static final long ***serialVersionUID***: *a unique identifier for Serializable classes*

  final private int maxSize: *maximum size of the queue.*

  Methods:

  public EvictingQueue(int size) : a constructor that instantiate **EvictingQueue**

  public boolean add(K k): add element on the queue and pop the oldest element if the queue reaches the maximum size.

❏ class StationsMap: extends java.lang.Object

StationMap class represents a map that holds the position of all stations and stores their states as well as the position of drone. It handles all functions related to calculating distances, locations of stations, directions, downloading and merging maps among others.

Inner Classes:

static public class Station : is inner class that represents Stations in the map.

static public class MapDate : in class that represent date of the map.

Fields:

private Map<String, Station> stations : *stores all stations in map with keys being stations' unique ids, and values being **Station** objects. Holds information about stations in the map*

final private String mapSource : *stores original Json string of the downloaded map.*

final MapDate mapDate : stores the date of the downloaded map.

Methods:

public StationsMap(String mapUrl): a constructor that instationate the StationsMap by download and loading the map.

private String loadMap(String mapUrl) : downloads the map and initializes stations member of StationsMap object

public List<String> getStationsWithIn(**Position** position, double range) : return all stations ids within a given range from that particular position

public List<String> getStationsWithIn(List<String> stations, **Position** position, double range): *an overload functions that return stations ids of all stations in given station list within a particular range from a given position*

public Station getStationById(String stationId): return a **Station** object given its id

public String getClosestStation(Position position): *Gets the closest station in all stations from that particular position*

public String getClosestStation(List<String> stations, **Position** position): overloaded functions *returns closest station from the given stations's ids*

public String getMapSource(): return the original Json string source of the map.

public List<String> getAllStations(): return all stations ids in the map;

public List<**Direction**> getPossibleMoves(Position position): return legal moves on the map given a particular position.

public static **Direction** getDirectionTo(**Position** from, **Position** to, List<Direction> directions): *return the best directions that that takes one from 'from' position to 'to' position.*

public static double calcDistance(Position pos1, Position pos2) : static functions that calculate the euclidean distance between two positions on the map.

public static String downlaodMap(String url): given a url it return json string of the map.

@throws *MalformedURLException the malformed URL exception*
@throws *IOException Signals that an I/O exception has occurred.*

public static String mergePathToMap(String mapSrouce, List<**Point**> path) : takes geoJson and list of points and create a LineString and adds it to the geoJson of the map.

public static String formatMapUrl(**StationsMap.MapDate** mapDate, String baseUrl, String geoJsonFile) : *format the URL on which to download the map*

- **class Station:** extends **java.lang.Object** and is an inner class of **StationMap**

represents a stations will all information and methods to access all those information related to that particular station.

Fields:

final private String id : a station id
final private Position position
private float coins
private float power

Methods: All methods are getters and setters of the member fields.

public Station(String id, float coins, float power, Position position)
public String getId()
public float getCoins()
public void setCoins(float coins)
public float getPower()
public void setPower(float power)
public Position getPosition()

- **class CommandArgsParser**: extends **java.lang.Object**

this class is used to parses command argument passed to the simulator and provide good APi to access information need by various classes.

Fields:

private MapDate mapDate;
private Position startCorridinate;
private int generator;
private String droneType;
private boolean isStateful;

Methods:

public CommandArgsParser(String args[]): *a constructor that takes all the command line arguments and call parser.*

private void parse(String args[]) : *parses all of the argument passed to the simulator and set all information parses to their corresponding fields.*

@throws *IllegalArgumentException the illegal argument exception*

- **class MapDate** : extends **java.lang.Object** and is inner class **of StationMap**

Holds the date of a particular map and provides functions to print it in the desired format.

Fields:

final private String day;
final private String month;
final private String year;

Methods:

public MapDate(String date,String mount, String year): a constructor that instantiate the date object.

public String toString() : returns all information of an object: da, month, year

public String formatDate(String delimiter, boolean reverse): takes a delimiter to separate the output date and boolean that determines either to return the date in reverse format or in day, month, year order.

- **final class IOUtils**: extends **java.lang.Object** and is a Utility class

  class IOUtils *is a utility class that contains all helper* static *functions used when were saving output files It mostly deals with IO and related formatting by prettifying* geoJson in user friendly readable format or formatting filenames.

  Methods:

  public static void saveJson(String jsonString, String filename) : *given a json string it saves it with $filename.geojson file by first prettying it.*

  public static void saveLogs(String logs, String filename): *given a filename, and logs string, it save the string in $*filename*.txt extension file of given string.*

  public static String prettifyJson(String jsonString): *prettify Json string to be saved in structured format.*

  public static String *generateFileName(**Drone** drone, **MapDate** mapDate)* : given a drone and a map date, it generates a standard filename that is used to save geoJson Map and .txt logs.

References

MapBox Library was used for parsing downloaded geoJson. More information about MapBox can be found on the following link:
https://docs.mapbox.com/android/api/mapbox-java/libjava-geojson/4.9.0/index.html?com/mapbox/

Visualization of GeoJson output:

- **http://geojson.io/**