ILP Report 2019

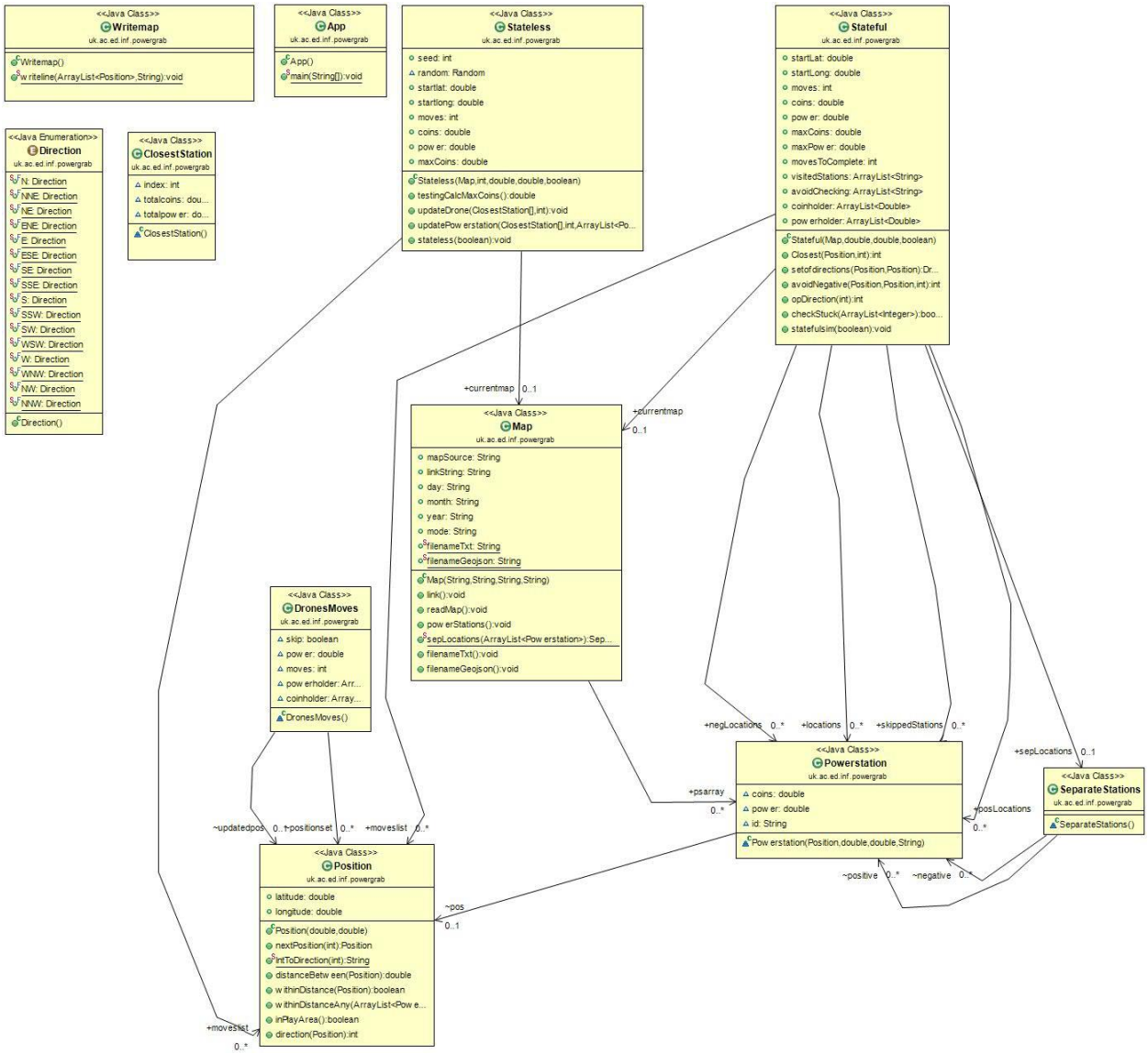
Sean Train s1740981

Table of Contents:

Software Architecture Description

UML Diagram:

The UML diagram below has been added to assist in the description of the software architecture:



Description:

The Position class was used to provide a way of holding the position of the drone and of the power stations on the map through their latitude and longitude. This is one of the most important classes as the entire project relies on using the position the drone and the power stations to determine the drones next move and where to avoid travelling. The Direction class holds the 16 directions that the drone can travel through an enum data structure which is important since the directions must be stored as constants. The

Map class is very important to the project through downloading and parsing the map from the informatics server. This is done through storing the information parsed into an arraylist of power stations which are defined in the Powerstation class that has a custom data type for storing the information of a power stations such as the id, coins, power and position which uses the Position class. This class is used in the App main method to parse and store the map data into the powerstation data structure before passing that the structure into the stateless and stateful drone classes for further use. The Stateless class is one of the main features of the project by providing the stateless drone implementation including the random number generator for direction, the strategy for choosing which direction to travel in and writing a text file for the stateless drone's moves. The stateless class uses the ClosestStation class which has a custom data type for storing the closest powerstation's data in relation to the drone's position and the direction travelling. This class is important since it simplifies storing the power stations information into one data structure instead of individual data structures each holding one part of the power stations information which would make the implementation less readable. The Stateful class is the most important class in the project through implementing the stateful drone implementation. This class makes use of the SeperateStations class which has custom data type of 2 arraylists for the positive and negative power stations which are used in the Map class's method for separating all the power stations into those that are positive and negative. Furthermore, the stateful class also makes use of a custom data structure held in the DronesMoves class that holds values such as the set of positions, new position of drone and skip status for the simulation of the drone travelling from its current position to a new powerstation. This is an important class since the way the strategy is executed is through using the data gathered from a series of moves from the current position of the drone to the next powerstation. Once the stateful drone has finished moving to the last powerstation, the values of that simulation can be extracted through the global variables that cover the coins the drone collected, the power the drone collected, the number of moves it took to complete the map and the moves list which will be used for writing the path of the drone to the geojson file. The final class Writemap is executed at the end of the main method in App and is important in the project through the creation of the geojson map of the initial map downloaded with the added LineString for the stateful or stateless drone's path depending on the mode passed to the class by the main method through the command line input/start run configuration.

Class Documentation:

Direction:
Description: The direction class contains an enum data structure that holds the values for all the 16 directions.
Methods: none.

Powerstation:
Description: The powerstation class is used as a custom data structure that stores the values that each powerstation holds and a constructor for creating a powerstation object. These values are id, coins, power and the position of the powerstation on the map in latitude and longitude.
Methods: none.

SeparateStations:
Description: The SeparateStations class is used as a custom data structure for holding 2 arraylists of power stations, one for positive stations and one for negative stations
Methods: none.

ClosestStation:
Description: The ClosestStation class is used as a custom data structure for holding the closest station's data such as the index position of the station, the total coins and the total power.
Methods: none.

DronesMoves:
Description: The DronesMoves class is used as a custom data structure for storing the set of positions, the final position, the status of skip when attempting to visit a powerstation, the sets holding coins and power after each move and also the power value and number of moves the drone had before it started moving to ensure that if the station is skipped, those values are restored in the *statefulsim* class.
Methods: none.

Position:
Description: The position class contains a data structure for storing the latitude and longitude and a constructor for creating a position object. Contains several methods for manipulating positions into new positions, directions and strings.
Methods:
Position nextPosition(int) - given an integer direction (between 0 and 15) will return an updated position object for the drone after moving in that given direction.

String IntToDirection(int) - passed an integer direction (between 0 and 15) and returns the string value of that of that direction. (0 = “N”, 1 = “NNE”, etc.)
double distanceBetween(Position) - passed a position and calculates the distance between the given position and the position object the method is called on using the Euclidean distance formula.
boolean withinDistance(Position) - passed a position and returns true if the position given and the position object the method is called on are within a 0.00025 distance of each other using the *distanceBetween* method within the same class.
boolean withinDistanceAny(ArrayList<Powerstation>) - passed an arraylist of power stations and returns true if there is one or more power stations that have a position that is within a 0.00025 distance of the position object the method is called on by using the *withinDistance* method in a loop over the arraylist of power stations.
boolean inPlayArea() - returns true if the position object the method is called on is within the given play area’s latitude and longitude borders.
int direction(Position) - passed in a position and calculates the degree of the straight line between the position object the method is called, and the given position has before converting that degree into an integer direction (between 0 and 15) and returning it.

Map:

Description: The map class stores the day, month, year and model of the command line input and uses this data to download the geojson map from the web server. The class also parses the data into a powerstation structure to make the data usable. The map class has several methods that use the passed in values to download, parse and store the geojson map.

Methods:

link() - uses the values of day, month and year passed into the map class to create a link to the geojson map for use in later methods.
readMap() - uses the global variable linkString populated by the *link()* method to open a URL connection to the webserver and reads the file line by line using StringBuilder before storing the entire map as a String mapSource.
powerStations() - uses the mapSource string created by *readMap()* and creates a list of features which are power stations. Loops through all the power stations and stores the position, coins, power and id of the power stations into the custom data type Powerstation and storing the power stations into an arraylist of power stations.
SeparateStations sepLocations(ArrayList<Powerstation>) - passed in an arraylist of all of the power stations calculated in *powerStations()*, the method separates the power stations into positive and negative power stations based on their coins value and are stored in a custom data structure SeparateStations which contains 2 arraylists, one for positive stations and one for negative stations.

filenameTxt() - uses day, month, year and mode passed into the map class and stores the stores the text file name in the global String variable filenameTxt.
filenameGeojson() - uses the day, month, year and mod passed into the map class and stores the geojson file name in the global String variable filenameGeojson.

Writemap:

Description: The Writemap class is used for creating the geojson file with the drone's flightpath on it.

Methods:

writeLine(ArrayList<Position>, String mapSource) - takes in the arraylist of positions that drone moved in and the mapSource geojson file that is parsed into a string by the Map class. The method prints a geojson file with containing the mapSource with the added lineString of the drone's flightpath.

Stateless:

Description: The Stateless class implements the stateless drone simulation through several methods that control the drone by visiting a positive powerstation if the drone happens to be in range in its next move, updates the drones values after charging and updates the power stations values after charging as well as prints the drones text file for each move. The class contains a constructor for instantiating the class, taking in parameters for the simulation such as the current geojson map, the random number seed, the starting position (latitude and longitude) of the drone and the boolean writefile that determines if the text file will be written for this simulation.

Methods:

updateDrone(ClosestStation[], int) - taking in the array of 16 closest stations (one for each direction) and the direction integer, the method will update the drones coins and power according to the coins and power values in ClosestStation[direction]. This method considers that the drones coins and power cannot go into negative and therefore checks this using the Math.abs function to find the absolute value of the negative powerstation's values so that they can be compared.
updatePowerstation(ClosestStation[], int, ArrayList<Powerstation>) - passing in the 16 closest stations (one for each direction), the direction integer and the arraylist of all power stations, the method will update the powerstation values that has the index value ClosestStation[direction].index according to the current value of the coins and power of the drone. The powerstation cannot go into negative in either it's coins or power values so this method checks to ensure that this cannot happen by comparing the power stations values to the drone's values and updating accordingly.
stateless(boolean) - this method runs the stateless implementation and takes a boolean value that is passed into the Stateless class to determine whether to print the text file for this simulation. This boolean variable is false when testing the stateless simulation on

all maps to improve efficiency, set to true when run on a single map. The stateless method uses the start latitude and longitude passed into the Stateless class as well as the arraylist of power stations that is passed in, created by the *Map.powerStations()* method. The method creates 2 arraylists that will store the directions that the drone is able to move (canGo) and not able to move in (cantGo) after each move and compares the coins value of the power stations that are held in the canGo arraylist and chooses the largest to move in. If all the coin values are the same, the drone will choose a seeded random integer to be used as an index value in the canGo arraylist that chooses where the drone will move to. The canGo and cantGo arraylists are cleared after each move the drone makes, ensuring the drone is stateless. The closest station index value if there is no station in range will be -1 and the *updateDrone* and *updatePowerstation* methods will only be called if the closest station index is not -1 (there is a station in range of the next move to charge from). Each move of the drone is written to a text file and the position of the drone after moving is added to an arraylist move list that is used to write the geojson file for the drone's flightpath. The simulation makes use of a do loop that will only exit after 250 moves by the drone or if the drone's power is below 1.25 units.

Stateful:

Description: The Stateful class implements the stateful simulation of the drone through several methods that find the closest of all/positive/skipped power stations from the current position of the drone, the set of values such as the positions and updated position from the drone travelling to a powerstation, directions that avoid negative stations or out of the play area positions and the most important method that uses these others methods to simulate the stateful drone's strategy for completing the map. The class contains a constructor for instantiating the stateful simulation taking in parameters for the specific run such as the current geojson map, the drone's starting position (latitude and longitude) and a boolean value actually charge that determines if this simulation is going to have the text file written for it and the powerstation values updated.

Methods:

int Closest(Position, int) - this method takes in the current position of the drone and the integer mode that determines which set of power stations the method will check the drones position to be closest to (mode = 0 is the positive stations, mode = 1 is all the stations and mode = 2 is the skipped stations). The method uses the specific global variable arraylist that is required depending on the mode integer to return the closest station's index value based on the position of the drone using the *Position.distanceBetween(Position)* method.

int opDirection(int) - this method takes in an integer direction and returns the opposite integer direction to the one given (for example: if direction was 0, the opposite direction would be 8).
boolean checkStuck(ArrayList<Integer>) - this method takes in an arraylist of directions that the drone has been moving in and checks if the drone travels in a sequence of cancelling directions using the *opDirection* method that result in the drone getting stuck by travelling back and forth. If this happens the method returns true else, it returns false.
int avoidNegative(Position, Position, int) - this method takes in the drone's current position, the position of a powerstation the drone is trying to visit and the current direction that the drone wants to travel in. This method then creates an arraylist for the directions the drone cannot go in (cantGo) by looping through the drone travelling one move in the 16 directions and if the going into a negative power station or out of the play area, that direction is added to the cantGo arraylist. The method then loops through all 16 directions and uses cantGo to determine the direction that would get the drone the closest to the powerstation it is trying to visit without going in one of the cantGo directions. If all 16 directions are in cantGo, the drone will travel in the original direction given to the method. Once the direction has been calculated, it is returned.
DronesMoves setofdirections(Position, Position) - this method takes in the values of the drone's position and the powerstation's position that the drone is trying to visit. This method finds the direction from the drone's position to the powerstation's position using the *Position.direction* method and then checks if that direction will cause the drone to go into a negative station and prevents this using the *avoidNegative* method. It will then update the drone's position by going in that direction and an arraylist for holding the position of the drone will have the new position added as well as an arraylist for directions will have travelled in direction added. 2 arraylists holding the coins and power values for that move are also updated after each move of the drone. The power and moves are also updated after each move the drone makes. The method checks to ensure that if the drone gets stuck (goes back and forth) when trying to visit the station through the *checkStuck* method then the skip value is set to true and the power and moves of the drone as reset to the value the drone had from the starting position that was passed into the method. The DronesMoves data would contain the skip status as true and the initial power and moves values for them to be reset in the *stateful* method. If the drone successfully visits the powerstation, it will return the DroneMoves custom data structure that contains the set of positions the drone went in, the set of power and coins each move had, the new position of the drone and the status of skip which would be false since the station was successfully visited.
statefulsim(int, boolean) - takes in a boolean variable actuallycharge that dictates if the text file for the simulation will be written. The closest powerstation is calculated using the *closest* method and then *setofdirections* is called cause the drone to visit this station and the DronesMoves data is stored in a variable temp. The skip value from the

DroneMoves temp variable is checked to see if the drone was able to visit the powerstation and if so, the drones current position is updated, the coins and power are updated, an arraylist positionset that stores all the positions the drone's moves is updated, the coin and power holders are updated with the values for each move as well as an array list for holding the visited stations has the station index of the closest station that has been visited added. If the station is skipped, it is added to a skipped stations arraylist with power and moves being set back to before the attempt to visit the station took place. The is repeated for the next closest station and so on until every station has been visited or skipped. After each attempt to visit a powerstation, the skipped stations are then attempted to be visited and if this succeeds the skipped station is removed from the arraylist but if it fails this station will continue to be checked until the end of the do loop where the drone has either ran out of power or moves left to make or has been able to visit the skipped stations with every attempt used. If all stations have been visited/skipped and there are still moves left over, the drone will go back and forth in the same 2 directions until all moves or power have run out. If the boolean variable actuallycharge passed into the method is true, then the text file for the stateful drone simulation will be written.

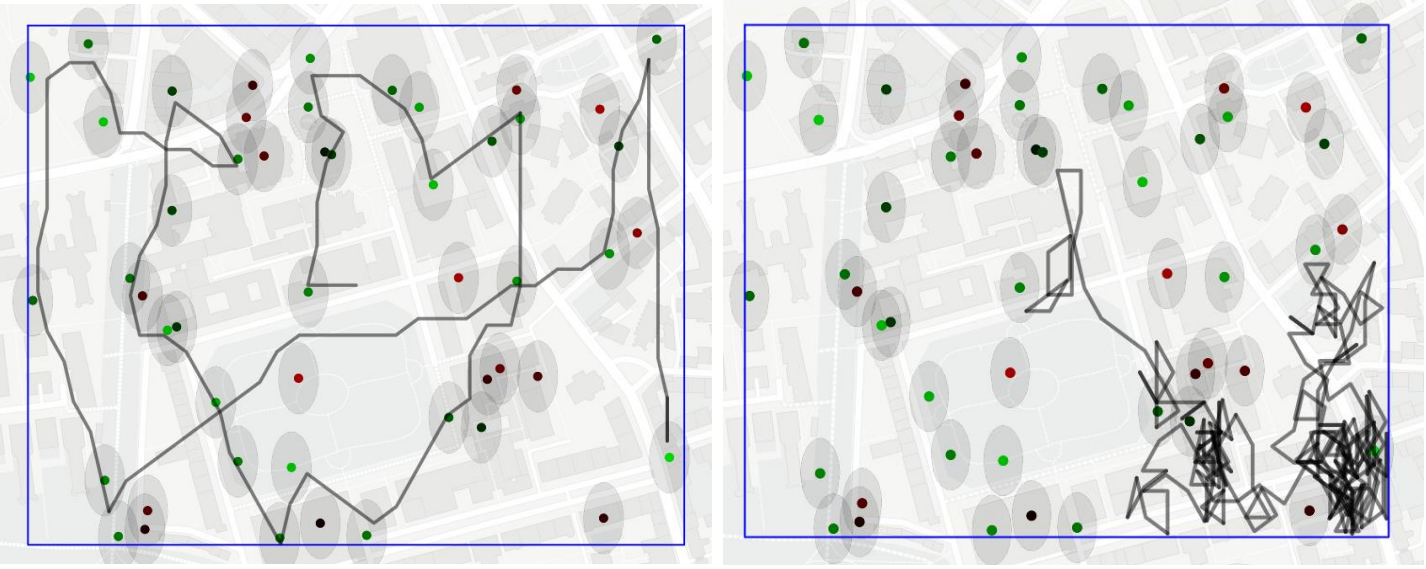
Stateful Drone Strategy:

The stateful drone will simulate visiting the closest powerstation station and if it is successful then the coins and power values are updated for the drone and the powerstation that the drone just visited has its coins and power set to 0 as well as the power stations id value being added to an arraylist called visitedStations that keeps track of all of the stations that have been visited. If it fails to visit this powerstation, then the power and moves of the drone are reset to the values before the drone started moving towards this station and this powerstation is added to an arraylist that stores all of the skipped power stations.
This is repeated within a loop for the drone to visit every other power station that is remaining by using the *closest* method to find the closest powerstation to the drone's current position. Once the drone simulates a visit to a powerstation that is unsuccessful, then the skipped powerstation arraylist will now have a powerstation in it and a statement at the top of the loop is constantly checking to see if there is at least one skipped station. If this statement returns true (skipped powerstation arraylist is not empty) then after each attempt to visit a new positive powerstation, the drone will also attempt to then visit the skipped power stations from its current position, attempting to travel to the closest skipped powerstation first. If successful, then the drone will charge from this powerstation and the powerstation is removed from the skipped power stations arraylist and added to the visited powerstation arraylist. If the drone is unable to charge from the skipped powerstation then it is added to avoid powerstation arraylist and then the next closest skipped powerstation is attempted to be visited if there is more than one. The reason that the failed-to-visit skipped powerstation is added to the avoid power stations arraylist is to prevent it showing up at the closest powerstation when the drone is wanting to visit the next skipped powerstation. After all the skipped power stations have been attempted, the avoid powerstation arraylist is cleared to ensure that after the next attempt to visit a non-skipped powerstation, the previously failed skipped powerstation can be attempted to be visited again. (Previously, I had set the stateful drone only try to visit the skipped power stations at the end of the loop after the drone had visited every other non-skipped powerstation. However, I found that this approach caused some power stations to be unable to be visited when trying at the end and the stateful drone would end up missing out coins and power making it less successful. Furthermore, it increased the efficiency of the drone in terms of moves made to visit all of the powerstation when checking to see if the drone could visit the skipped stations after every attempt at a non-skipped station. This was because it would require less moves to visit the skipped station after just visiting a nearby powerstation since the stateful drone works by travelling to the closest station each time and it was found that the skipped powerstation was able to be visited usually after 1 or 2 attempts from another powerstation reducing the overall moves of the drone. Once the drone has

visited all the positive stations or there are some skipped stations that were never able to be visited (this is rare occurrence), the drone will repeat the last 2 moves back and forth until all 250 moves are used up. The way the loop for simulating the remaining moves is done by choosing the last move of the drone where it visited the last powerstation successfully and then getting the opposite for this so that you have 2 positions that are one move apart from each other and are safe from going into negative stations or out of the map due to being previous moves the drone has made. From this, a loop is used to make the drone travel back in forth between these 2 positions until all 250 moves have been used up. Once the stateful drone has made all 250 moves, the variables holding the coins and power for each move as well as the moves list hold all the moves made by the drone are used to print each line of the stateful drone's text file.

A comparison between the performance of the stateful and stateless drone has been shown below with the help of 2 images for the drones being run on the same map:

The input for the drones is the same "*04 04 2020 55.944425 -3.188396 5678 drone*" where drone is type of drone. Left image is stateful, right image is stateless.



To improve the stateful drone over the stateless drone several changes were made including allowing the stateful drone to simulate travelling to a powerstation and then deciding whether to travel there depending if the drone was able to successfully charge from it. Another change that was implemented for the stateful drone is that is can look ahead all the moves it requires to visit the closest power station from its current location whereas the stateless drone is only allowed to look ahead one move.
Furthermore, the stateful drone has no elements of randomness, instead being controlled based on data produced from simulating a visit to the nearest power station.

The stateful drone has a strategy for visiting the closest positive powerstation which is much more efficient than visiting a powerstation based on a random number generator that decides which move to make and can only enter a station if the stateless drone is within 1 move unlike the stateful drone which could theoretically visit a station that is thousands of moves away if the stateful drone was not restricted by the map size, maximum number of moves and power levels.
In each state, the stateful drone has access all the positive powerstations left, the skipped powerstations that need to be checked, the avoid powerstations that don't need to be checked on this move, all the stations that the drone has already visited and charged from and its coins and power values before and after every move it makes. All of this is used to ensure that the drone can visit the closest station to its position before checking the skipped stations and overall, attempting to visit every positive powerstation on the given map.

From the testing code that is within App (setting the variable all to "yes" and year to either "2019" or "2020), the results for the accuracy are shown below *:

Drone	Year	Accuracy (%)
Stateful	2019	99.94735742042744
	2020	99.97488818420968
Stateless (5678)	2019	40.44963744896405
	2020	40.85515896381464
Stateless (100)	2019	46.97061189454873
	2020	48.232669040172475

*the maps were between the 1st and 28th (inclusive) of every month of the specified year.

From these results, the stateful drone is far superior to the stateless drone due to the reasons that have been stated above. I was surprised to see that the stateless drone was able to accumulate roughly 40% average over all the maps with the random seed of 5678 and an average of around 47% when using a random seed of 100. The seed for the stateless drone impacts the results to an extent whereas the strategy for stateful drone is reason for such a high accuracy over all maps which is due to the strategy not containing any randomness.