CW-model : Implementation of MyGameStateFactory(82/82 tests)

The first tasks were relatively straightforward writing the first getter methods for MyGameState, checking for illegal arguments in the constructor as well as defining and initializing atributes (i.e: information about a given game state). There are only a few minor things I want to point out in this section:

- In the constructor of my MyGameState, I also added an extra integer argument "round". This is because we need to keep track of the current round number in some ways to implement more complicated methods such as the advance method.
- I have also added helper functions: CheckMrX, CheckMoreThanOneMrX, CheckDuplicateDetectives, CheckLocationOverlapBetweenDetectives, CheckTicketsDetectives, CheckEmptyRoundAndGraph. These functions are just arguments checkers but I made a seperate function for each of them to make the code look more compact and readable.
- In getPlayerTickets, I realized that the Ticketboard interface has only 1 function so a lambda funciton notation could be applied here. (`return Optional.of(ticket -> p.tickets().get(ticket));`). This is not much but it was one of the things that I am most proud of in this coursework since Anonymous inner class and lambda functions were so confusing and untuitive to me at the time and I could finally grasp them.

GetWinner And Get AvailbleMoves:

These are more complicated methods as moves and winner cannot be easility assembled. Just like in the guide, I had GetAvailableMoves return moves whose calulation is done in the constructor of MyGameState. There are 2 helper functions: makeSingleMoves and makeDoubleMoves.

- makeSingleMoves: a function that calculates all single move a player can make given his location and the current gamestate. The logic was to loop through all the adjacentNodes to the source (I.e: player's location), Check for availability (not already occupied by a detective and player has the required tickets, which can also be a SECRET ticket). One thing to note was the edgevalue of this graph between node A and node B is the transport "t" that connects note A and B. So t.required ticket can be used when checking if player has the required ticket(s).

- makeDoubleMoves: a function that calculates all double move a player can make given his location and the current gamstate. Only mrX can make double moves. The logic was to make a nested for loop, one for all the adjacent nodes to the source and then one for all the adjacent nodes to each of those nodes, each time checking for their availability. One thing to note was that we need to check if the player still has the required number of tickets after using one for the first move. (e.g: Does he have 2 bus tickets if he want to use the bus twice). I made a MutableCopy of the player's ticket and substract the used ticket by 1 after constucting the first move.
- Checkwinner is a helper function to check if there is a winner when a gamestate is constructed. This value is returned in the getter method GetWinner.

<u>Advance Method:</u>

My idea was to update every arguments of the constructor of gamestate and construct a new gamestate, which is returned by the advance method, with the updated arguments. For each of the arguments needed, I wrote a helper function

- UpdatePlayer. This function returns a new player, whose location will be the move final destination. (I.d move.destination for SingleMove and move.destination2 for DoubleMove)
- UpdateLog. This function return a new mrX travel log. It behaves differently for different types of moves so Visitor pattern was applied
- GetNewRemaining. This function updates the List of the player that has not yet made the move in the current round.

We would then use these updated values to construct a new Gamestate. One thing to note was that round needed to be incremented if it was the start of the next round.

CW-AI: Implementation of the Dijkstra's shortest path algorithms.

Foreach of the available moves, I associated it with a score, I will then return the move with maximum score using the "max" help function I created. The shortest paths from the move destination to every detectives' locations are calculated using the

Dijsktra’s algorithm. These distances would then be averaged to give the score of that move.

- Score(Move move, Board board)

  This function calculates and asscociate a move with a score. It will behave differently for singleMoves and DoubleMoves so visitor pattern was applied

  Dijsktra(map, move.destination)

  Foreach of the move. The shortest paths from the move destination to every detective locations are calculated using Dijsktra’s algorithm and stored in “distance”

  The Score function would then return SumofDist / detectives.size which is just the average of all the distances calculated.

- Dijsktra(Map, source)

Nodes is a list of all nodes in the graph

dist is our map to return. Which basically maps nodes to corresponding Dijkstra distances

First I initialized dist of everynode to be 999999.0 except for the source node being 0.

The algorithm starts at the node with the minimum value in dist then loop through every adjacent node and replace the corresponding dist value with a smaller value if its found.

The weight(node, adjacentNode) function only returns 1 as a starting point because it makes sense to have distance between any 2 adjacent nodes to be 1. However, it could be modified so that it would take into account the transport needed to travel between those nodes as well as the player’s ticket and average those values out with some functions.