CE811 Assignment 1

Student registration number: 2312447

1. DFS maze generator.

```
def build_maze_grid_dfs(n_rows, n_columns):
    # Build a grid of maze-cells, each initially populated by walls
    grid = Grid(n_rows*2+1,n_columns*2+1) # We double the sizes here to make space for walls surrounding each proper walkable cell.
    # Now run DFS algorithm which should carve out the pathways.
    current_node = (1,1) # always starts from one-step diagonally inwards from top-left corner
    stack =  [current_node]
    while len(stack)>0:
        current_node = stack.pop()# Get highest priority cell from stack (Depth-first search)
        neighbours = grid.get_unvisited_neighbours(current_node)
        print("Current Node",current_node, "Neighbours",neighbours)
        #TODO finish this code block here to complete the algorithm!
        if len(neighbours)>0:
            neighbour = random.choice(neighbours)
            grid.remove_wall(current_node, neighbour)
            grid.set_visited(neighbour)
            stack.append(neighbour)
        else:
            if len(stack)>1:
                stack.pop()
            else:
                break
    return grid
```

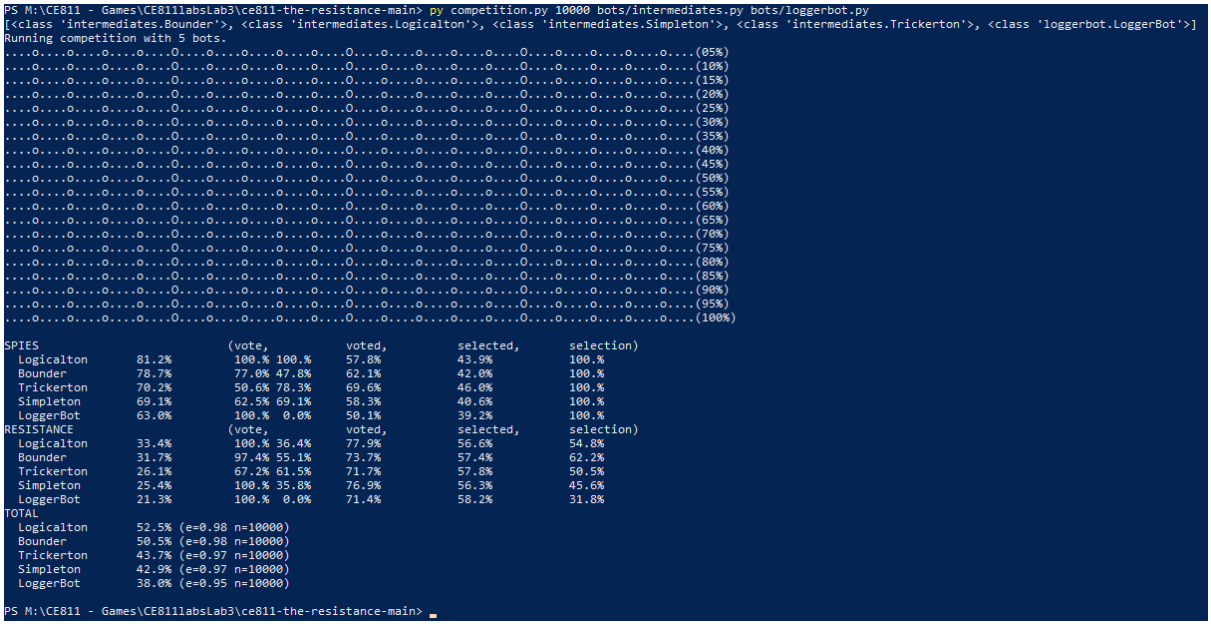
From the comment saying TODO is my code. Within this code I was asked to complete the algorithm for building a maze. If we look at this line by line I will explain what is happening. The first if statement is firstly finding if the length of the neighbours variable is greater than 0. This is to see if there are any unvisited neighbours. If there are unvisited neighbours then we randomly pick one from the list. The next line will remove the wall between the current neighbour and the randomly chosen neighbour so that the maze can develop further. After the wall is removed we mark the chosen neighbour so that we can keep track of what is happening and what has been explored. The chosen neighbour is then added to the stack so that the algorithm can keep exploring until it hits the same issue, and we backtrack. We also have to add an else in case the list has no values in it. If this is the case, we pop the top element, causing some backtracking, this way we can explore more of the maze. If the stack has only one element we break, as we would have explored the whole maze. This also terminates the loop, stopping a possible infinite loop.

2. NeuralBot for The Resistance.

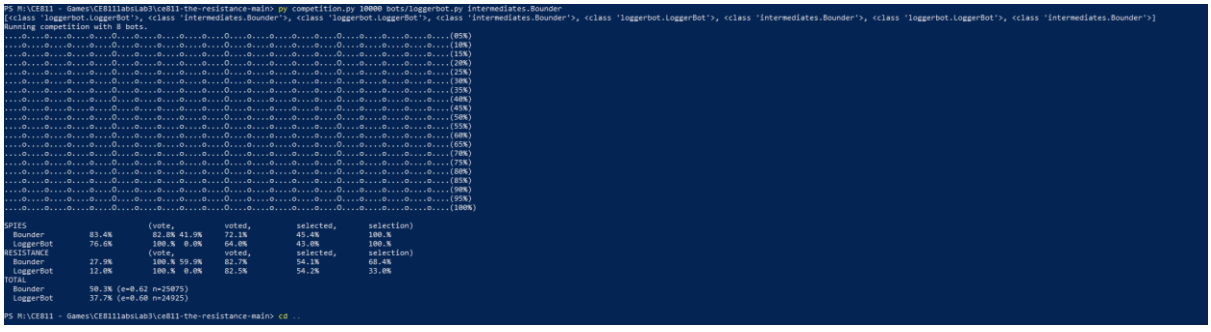
There is one primary input with that being the calc_player_probabilities_of_being_spy. Which will feed into the neural network and will use multiple methods to figure out the outputs, in which there are 4. The input however is made up of multiple input vectors which all help train the neural network. These are the game turn variable which tells us the current game turn. The missions been on, which tells us the missions that we have been on. The missions failed on, which tells us the number of missions that we have been on and that have failed. The num_missions_voted_up_with_total_suspect_count, this tells us the number of missions in which we voted up on with the total number of suspect count. Which is a list that counts our suspects for us. This list will change depending on if the mission is a success or not. Lastly the num_missions_voted_down_with_total_suspect_count, which does the same as the last variable however counts the missions voted down not up.

The outputs are the calc_player_probabilities_of_being_spy method, this will return a dictionary of players names which will be linked to there probability of them being aa spy. The select function which is a list of players that have been selected for the mission organised in their trustfulness, to complete a mission or not. The vote which will return a boolean on whether we should allow the mission to pass or not. The sabotage, which again will return a boolean to let us know if this mission has been sabotaged or not, although this is not a direct output from the neural network it's worth noting.

Screenshot vs 1000 bots(played 10000 times)



Screenshot vs bounder and logger (played 10000 times)



3. Connect-4 Static Evaluator

The static evaluator is mainly used to evaluate the game board at specific phases of the game. The method takes two parameters *'board'*, which represents the game board and *'piece'*, which represents the player's piece. We define the opponent's piece by taking the board grid and seeing if a 1 or 2 is placed there, if 1 its our piece if 2 it's the opponent's piece. We also have the score set to 0 so we can increase it based on some conditions. We then go through some if statements which will check for potential winning combinations for the current player's piece. It will check horizontally, vertically and diagonally. We also use the evaluate_window method which helps find the winning combinations in the if statements. It does this by having using a score, which is set to 0, based on the number of player pieces and empty slots. This allows us to make better decision making while playing connect 4. The main approach of the static evaluator is a simple form of heuristic- based evaluation.

```python
def static_evaluator(board, piece):
    grid = board.grid
    opponent_piece = 1 if piece == 2 else 2  # Determine opponent's piece
    score = 0

    # Check for wins
    for row in range(6):
        for col in range(4):
            window = list(grid[row, col:col + 4])
            score += evaluate_window(window, piece)

    for col in range(7):
        for row in range(3):
            window = list(grid[row:row + 4, col])
            score += evaluate_window(window, piece)

    for row in range(3):
        for col in range(4):
            window = [grid[row + i, col + i] for i in range(4)]
            score += evaluate_window(window, piece)

    for row in range(3):
        for col in range(3, 7):
            window = [grid[row + i, col - i] for i in range(4)]
            score += evaluate_window(window, piece)

    return score

def evaluate_window(window, piece):
    opponent_piece = 1 if piece == 2 else 2

    # Score the window based on the number of pieces and empty slots
    score = 0
    if window.count(piece) == 4:
        score += 100
    elif window.count(piece) == 3 and window.count(0) == 1:
        score += 5
    elif window.count(piece) == 2 and window.count(0) == 2:
        score += 2
    if window.count(opponent_piece) == 3 and window.count(0) == 1:
        score -= 4
    return score # TODO enhance this logic so that your static evaluator gives useful recommendations.
```

4. Connect-4 Minimax algorithm

```
def minimax(board, current_depth, max_depth, player, alpha=-math.inf, beta=math.inf):
    if player == board.get_player_turn():
        maximiser = True
    else:
        maximiser = False

    is_terminal = board.is_game_over()
    if current_depth == max_depth or is_terminal:
        if is_terminal:
            opponent = 3 - player
            if board.get_victorious_player() == player:
                return (None, +100000000)
            elif board.get_victorious_player() == opponent:
                return (None, -100000000)
            else:
                return (None, 0)
        else:
            return (None, static_evaluator(board, player))
```

```
    valid_moves = board.valid_moves()
    if maximiser:
        best_move = None
        best_value = -math.inf
        for move in valid_moves:
            new_board = board.play(move)
            _, value = minimax(new_board, current_depth + 1, max_depth, player, alpha, beta)

            if value > best_value:
                best_value = value
                best_move = move
            alpha = max(alpha, best_value)
            if alpha >= beta:
                break
        return best_move, best_value
```

Here we have added the alpha-beta pruning, as see in the images above and below. In the maximiser part we initialize best move and best values. We then start the loop for maximiser which is where we will do our alpha pruning. The alpha pruning is done by checking if the pruning conditions have been met *'if alpha >= beta: break'*. If the condition has been met then, we break the loop as it is not beneficial for the branch to continue and explore unpromising branches. If alpha is equal to or greater than beta, then the condition is true, and we know that we do not need to consider this branch anymore as the minimizer will never allow this branch to be chosen. By doing this we stop unpromising paths from being explored, reducing search space and saving time. Beta pruning is the same as alpha pruning except we are checking if beta is less than or equal to alpha.

```
        return best_move, best_value
    else:  # Minimising player
        best_move = None
        best_value = math.inf
        for move in valid_moves:
            new_board = board.play(move)
            _, value = minimax(new_board, current_depth + 1, max_depth, player, alpha, beta)

            if value < best_value:
                best_value = value
                best_move = move
            beta = min(beta, best_value)
            if beta <= alpha:
                break
        return best_move, best_value
```

5. Connect-4 MCTS algorithm

To conduct this experiment, I set out a testing parameter that we will play both algorithms against each other 100 times. 50 times with Minimax being red player (making the first move) and, 50 times with MCTS being red player (making the first move). We will then compare the results of the experiment and see which algorithm performed the best.

Below is a graph of the results collected:

Times played	MCTS	Minimax	Draw
50 with minimax as player 1	33	17	0
50 with MCTS as player 1	23	27	0

Please note that Minimax is using alpha-beta pruning as well.

Let's break down the experiment results. As we can see when the minimax algorithm was player 1 it had a win rate of 17/50 or 37% and the MCTS algorithm had a win rate of 33/50 or 66%.

When we look at these results, we would assume that the MCTS algorithm is the better algorithm as it has the higher win rate however this seems to just be when it is playing second to the minimax algorithm.

If we now look at the second part of the experiment, where MCTS is player 1. We notice that its win rate drops to 23/50 or 46%. With the minimax algorithm increase to 27/50 or 54% surpassing the MCTS algorithm, after 50 games. This leads me to the following assumption, both these algorithms are better when playing second, after a move has been made but why is this?

My assumption would be that once a move has been played it allows for a lot more possibilities for the algorithms to analysis leading to better winning situations.

You will also see in the results that there was never a draw. I hypothesize that this is because when watching the games get played the algorithms always stacked on top of each other which is normal to see when humans play. I believe they do this as they are always looking forward to end of game therefore setting up the endings to be won with a single line remaining.

Another I noticed was that the minimax algorithm missed a lot of opportunities to win the game while it was player 1. There was at least 6 times that the minimax algorithm could have won the game but didn't due to choosing to play for the later game rather than just winning. This could have affected the results.

The last observation I noticed was that algorithms would always start in the bottom left of the grid if playing first. I hypothesize that this is due to not having any other information on the oppositions placement as they haven't placed anything yet.

To conclude I believe that the best algorithm out of the two above to use in connect 4 would be MCTS. This is based of my results above given that overall, it has a 56% win rate which isn't an overwhelming win rate however is if you are using the MCTS as player two it should win 66% of the time. Even if it is player 1 then it still has a 46% win rate which close to 50% in the context of the experiment that we conducted. There are ways to make the models more accurate by adding different layers to the minimax to make it better at the game.