Name: Ahmad Farhan

Roll no.i211366

Section: A

Assignment 3

Question 1: *Minimax TicTacToe*

Algorithm Logic

This algorithm essentially builds a game tree covering all possible moves from the current state to the end of the game and navigates this tree using depth-first search to find the optimal move. The minimax part of the algorithm ensures that the move chosen is the best one assuming an optimal opponent.

Algorithm Steps

1. Iterate over all empty cells on board, play those positions and call minimax function.
2. Minimax recursively simulates playing all the possible moves from current board state.
3. At each iteration we evaluate the desirability of current state based on all possible future moves
    - Maximizer tries to obtain maximum evaluated value from current state.
    - Minimizer tries to obtain the minimum evaluated value from current state.
4. Minimax terminates if no more moves can be played or either player wins.
5. Based on termination condition, calculate the desirability value using evaluate() function.
6. Returns best possible move with its desirability value for the player from current board state.

Part 1: *Simple Alpha-Beta (αβ-Minimax)*

Algorithm Logic

The αβ pruning algorithm is used to skip over branches of the game tree that do not need to be explored because they could never result in a better move than the current best move found. This significantly speeds up the algorithm by reducing the number of nodes it needs to explore in the game tree.

Algorithm Steps

1. Iterate over all empty cells on board, play those positions and call αβ-Minimax function.
2. αβ-Minimax recursively simulates playing all the possible moves from current board state.
3. At each iteration we evaluate the desirability of current state based on all possible future moves
    - Maximizer updates value of alpha and prunes further moves if alpha >= beta
    - Minimizer updates value of beta and prunes further moves if beta <= alpha
4. Minimax terminates if no more moves can be played or either player wins.

Algorithm Results

```
  Testcase: 1
['x', 'o', 'x']
['_', '_', '_']
['o', '_', '_']
```

```
  Testcase: 2
['_', 'o', '_']
['o', '_', 'x']
['_', '_', '_']
```

```
  Testcase: 3
['x', 'o', '_']
['_', '_', '_']
['_', '_', '_']
```

	Without Alpha-Beta Pruning				With Alpha-Beta Pruning			
No. of Nodes	Run1	Run2	Run3	Average	Run1	Run2	Run3	Average
	257	1420	8231	3303	183	673	1707	854

Part 2: *Parallel Alpha-Beta (αβ-Minimax)*

Algorithm Logic

The Aβ pruning needs to explore the tree sequentially to consistently prune the maximum number of branches. Note that the refined αβ values from earlier explorations prune later moves more significantly. However, the pruning in one root path from αβ value of previous root path is relatively less as compared to the pruning effect between the first and last root paths. Therefore, αβ minimax may be parallelized by executing n-adjacent root paths simultaneously and then updating the αβ values from the results of the parallel executions. Repeated destruction and creation of threads can introduce significant overheads; therefore, we will use a ThreadPoolExecutor from concurrent.futures module.

Note: that pruning effect will be less significant if the moves-to-threads ratio is high and the speedup factor will be less if the ratio is low.

Algorithm Steps

1. Create a ThreadPoolExecutor with max_workers = 2 using concurrent.futures module.
2. Iterate over two root moves at once and submit them with αβ-Minimax function to the executor.
3. The executor will run two threads αβ-Minimax function with different moves simultaneously.
4. The results of the two threads are used to update the αβ and best move values at root.
5. Updated αβ values at root are used in following iterations to retain maximum parallel pruning.

Algorithm Results

	Serial Alpha-Beta Pruning				Parallel Alpha-Beta Pruning			
No. of Nodes	Run1	Run2	Run3	Average	Run1	Run2	Run3	Average
	183	673	1707	854	183	606	1787	859
Time Taken (ms)	5.52	1.14	13.88	6.84	2.51	0.10	7.54	3.38

Testcase 3 Executions:
Note: No. of nodes evaluated is same i.e., serial(1707) and parallel(1787)

	Serial Alpha-Beta Pruning				Parallel Alpha-Beta Pruning			
	Run1	Run2	Run3	Average	Run1	Run2	Run3	Average
Time Taken (ms)	8.10	8.95	14.66	10.57	8.44	1.46	9.29	6.39

Part 3: *Heuristic Alpha-Beta (αβ-Minimax)*

Algorithm Logic

The αβ pruning typically explores all possible subsequent moves from current state in a random order. Heuristics based ordering of exploration from non-terminal states can lead to more effective pruning. During the search process, we use a heuristic to estimate the desirability of each of the following move states. Instead of exploring all possible moves in random order, we use the heuristic to prioritize which branches of the search tree are more promising and should be explored first. The ordering leads to faster convergence of αβ window.

Algorithm Steps

1. In αβ-minimax, before exploring the moves from current state, calculate heuristic of each move.
2. Maximizer sorts moves in descending order and minimizer sorts moves in ascending order.
3. Heuristic value of a move is the sum of score of all 3 consecutive cells (rows, columns, and diagonals) and is defined as a 2d matrix:
```
[   -1,   -10,  -100, -1000],
[   10,     0,     0,     0],
[  100,     0,     0,     0],
[ 1000,     0,     0,     0]
```
   Where the row is no. of player tokens (x) in line and col is no. of opponent(o) tokens in line.

Algorithm Results

```
  Testcase: 1
['x', 'o', 'x']
['_', '_', '_']
['o', '_', '_']
```

```
  Testcase: 2
['_', 'o', '_']
['o', '_', 'x']
['_', '_', '_']
```

```
  Testcase: 3
['x', 'o', '_']
['_', '_', '_']
['_', '_', '_']
```

	Simple Alpha-Beta Pruning				Heuristic Alpha-Beta Pruning			
No. of Nodes	Run1	Run2	Run3	Average	Run1	Run2	Run3	Average
	183	673	1707	854	51	307	261	206

Question 4: *Weak TicTacToe AI*

Steps

a) Unusable Tile:

1. Use np.random.randint(0,3) to generate a random position within board.
2. Use possible_moves.remove(unusable_move) to not let AI pick that tile.

b) Heuristic Score:

1. Use above defined heuristic function to evaluate desirability of each move.
2. Heuristic value of a move is the sum of score of all 3 consecutive cells (rows, columns, and diagonals) and is defined as a 2d matrix:

```
[    -1,   -10,  -100, -1000],
[    10,     0,     0,     0],
[   100,     0,     0,     0],
[  1000,     0,     0,     0]
```

   Where the row is no. of player tokens (x) in line and col is no. of opponent(o) tokens in line.

c) Random Selection of AI Move:

1. Determine all possible moves that AI can make, given a board state.
2. If only the unusable tile is left, then AI will pick that tile.
3. If only one usable tile remains, then AI will pick that tile.
4. Calculate heuristic scores of all possible moves and calculate total score of all moves.
5. Divide each score by total score to get heuristic probability.
6. Use np.random.choice to choose the move based on those probabilities.

Results

```
Enter Tokens as (user/ai): ox
Tokens Chosen: You are o, Ai is x
Enter to play 1(st) or 2(nd) :2
Ai will not use:  (2, 2)
['_' '_' '_']
['_' '_' '_']
['_' '_' '_']

Ai Played: (0, 0)
['x' '_' '_']
['_' '_' '_']
['_' '_' '_']

Enter your move(0..2,0..2): 01
You played:  (0, 1)
['x' 'o' '_']
['_' '_' '_']
['_' '_' '_']
```