ARTIFICIAL INTELLIGENCE

Assignment # 3

Deadline: Sunday, 21st April, 2024, 11:59PM

Instructions:

1. Assignments are to be done individually. You must complete this assignment by yourself. You cannot work with anyone else in the class or with someone outside of the class.
2. You have to use python programming language. You must attempt all parts.
3. Plagiarism of any kind (copying from others and copying from the internet, etc.,) is not allowed and can result in zero marks in whole assignment category.
4. **Your code must be properly commented.**
5. No marks will be assigned if any of the following deliverables are missing.
    a. The source code of the program.
    b. A pdf or word report containing a brief explanation of the steps involved in the program (each question) and the results obtained.
6. Put both source code and report in one folder, ZIP it and submit it. Your folder must be named as ROLLNO_NAME.ZIP. Late submissions will not be accepted.

Q1: Game playing is an essential part of AI and is the basis for many search techniques. Adversarial search refers to problems in which the agents' goals are in conflicts and may times refer to zero sum games where the win for one is the loss for the other. Minimax algorithm tries to find the optimal moves assuming that both the players play optimally.

https://www.geeksforgeeks.org/finding-optimal-move-in-tic-tac-toe-using-minimax-algorithm-in-game-theory/

Consider a scenario where tic-tac-toe is being played and the optimal move is found using minimax algorithm. The following code in python would do that:

```
# Python3 program to find the next optimal move for a player
player, opponent = 'x', 'o'

# This function returns true if there are moves
# remaining on the board. It returns false if
# there are no moves left to play.
def isMovesLeft(board) :

    for i in range(3) :
        for j in range(3) :
            if (board[i][j] == '_') :
                return True
    return False

# This is the evaluation function as discussed
# in the previous article ( http://goo.gl/sJgv68 )
def evaluate(b) :

    # Checking for Rows for X or O victory.
```

```python
    for row in range(3) :
        if (b[row][0] == b[row][1] and b[row][1] == b[row][2]) :
            if (b[row][0] == player) :
                return 10
            elif (b[row][0] == opponent) :
                return -10

    # Checking for Columns for X or O victory.
    for col in range(3) :

        if (b[0][col] == b[1][col] and b[1][col] == b[2][col]) :

            if (b[0][col] == player) :
                return 10
            elif (b[0][col] == opponent) :
                return -10

    # Checking for Diagonals for X or O victory.
    if (b[0][0] == b[1][1] and b[1][1] == b[2][2]) :

        if (b[0][0] == player) :
            return 10
        elif (b[0][0] == opponent) :
            return -10

    if (b[0][2] == b[1][1] and b[1][1] == b[2][0]) :

        if (b[0][2] == player) :
            return 10
        elif (b[0][2] == opponent) :
            return -10

    # Else if none of them have won then return 0
    return 0

# This is the minimax function. It considers all
# the possible ways the game can go and returns
# the value of the board
def minimax(board, depth, isMax) :
    score = evaluate(board)

    # If Maximizer has won the game return his/her
    # evaluated score
    if (score == 10) :
        return score

    # If Minimizer has won the game return his/her
    # evaluated score
    if (score == -10) :
        return score

    # If there are no more moves and no winner then
    # it is a tie
    if (isMovesLeft(board) == False) :
        return 0

    # If this maximizer's move
```

```python
    if (isMax) :
        best = -1000

        # Traverse all cells
        for i in range(3) :
            for j in range(3) :

                # Check if cell is empty
                if (board[i][j]=='_') :

                    # Make the move
                    board[i][j] = player

                    # Call minimax recursively and choose
                    # the maximum value
                    best = max( best, minimax(board,
                                              depth + 1,
                                              not isMax) )

                    # Undo the move
                    board[i][j] = '_'
        return best

    # If this minimizer's move
    else :
        best = 1000

        # Traverse all cells
        for i in range(3) :
            for j in range(3) :

                # Check if cell is empty
                if (board[i][j] == '_') :

                    # Make the move
                    board[i][j] = opponent

                    # Call minimax recursively and choose
                    # the minimum value
                    best = min(best, minimax(board, depth + 1, not isMax))

                    # Undo the move
                    board[i][j] = '_'
        return best

# This will return the best possible move for the player
def findBestMove(board) :
    bestVal = -1000
    bestMove = (-1, -1)

    # Traverse all cells, evaluate minimax function for
    # all empty cells. And return the cell with optimal
    # value.
    for i in range(3) :
        for j in range(3) :
```

```
                # Check if cell is empty
                if (board[i][j] == '_') :

                    # Make the move
                    board[i][j] = player

                    # compute evaluation function for this
                    # move.
                    moveVal = minimax(board, 0, False)

                    # Undo the move
                    board[i][j] = '_'

                    # If the value of the current move is
                    # more than the best value, then update
                    # best/
                    if (moveVal > bestVal) :
                        bestMove = (i, j)
                        bestVal = moveVal

    print("The value of the best Move is :", bestVal)
    print()
    return bestMove
# Driver code
board = [
    [ 'x', 'o', 'x' ],
    [ 'o', 'o', 'x' ],
    [ '_', '_', '_' ]
]

bestMove = findBestMove(board)

print("The Optimal Move is :")
print("ROW:", bestMove[0], " COL:", bestMove[1])

# This code is contributed by divyesh072019
```

Take this code and implement alpha-beta pruning to decreases the size of the search space by ignoring those subgraphs that may not lead to an optimal move.

Q2: Note, for each of the following:

1) **Assume a random starting position.**
2) **Perform 3 runs of the code.**

Perform the following evaluations, using the *code in the previous question:*

1) Calculate the average number of nodes that would be evaluated using the minimax algorithm without pruning and compare it to the number of nodes evaluated with alpha-beta pruning.

	Without Alpha-Beta Pruning				With Alpha-Beta Pruning			
No.of nodes evaluated	Run1	Run2	Run3	Average	Run1	Run2	Run3	Average

2) Implement a parallel version (2 threads) of alpha-beta pruning and compare its performance with sequential version.

	Parallelized Alpha-Beta Pruning				Serial Alpha-Beta Pruning			
No.of nodes evaluated	Run1	Run2	Run3	Average	Run1	Run2	Run3	Average
Time Taken								

3) Develop a heuristic for non-terminal states that would assist in reducing the search space (enhanced alpha-beta pruning) and compare it to the standard alpha-beta pruning algorithm.

Hint: *During the search process, when evaluating non-terminal states, use the heuristic to estimate the desirability of each state. Instead of exploring all possible moves to determine the exact value of a state, use the heuristic to prioritize which branches of the search tree are more promising. This involves comparing heuristic values and using them to guide the selection of moves and pruning of branches.*

	Alpha-Beta Pruning				Enhanced Alpha-Beta Pruning			
No.of nodes evaluated	Run1	Run2	Run3	Average	Run1	Run2	Run3	Average

4) Develop a tic-tac-toe game, where the AI is a “weak” player. Modify the game in the following manner:

   a) The “weak” player does not choose the best decision nor the worst decision but instead relies on randomization.
   b) Randomly pick a tile at the start of the game, which the “weak” player will not use during the game.
   c) Use a heuristic to define the desirability of a state.