VIT Vidyalankar Institute of Technology Accredited A+ by NAAC	**DEPARTMENT OF COMPUTER ENGINEERING**

Experiment No. 7

Semester	T.E. Semester VI
Subject	ARTIFICIAL INTELLIGENCE (CSL 604)
Subject Professor In-charge	Prof. Avinash Shrivas
Assisting Teachers	Prof. Avinash Shrivas

Student Name	Deep Salunkhe
Roll Number	21102A0014
Lab Number	310A

Title:

Tic tac toe using minmax algo

Theory:

1. Introduction to Tic-Tac-Toe:

Tic-Tac-Toe is a classic two-player game played on a 3x3 grid. The objective of the game is for a player to place their symbol (X or O) in a row, column, or diagonal, aiming to achieve three consecutive symbols and win the game.

2. Minimax Algorithm:

The Minimax algorithm is a decision-making algorithm commonly employed in two-player games with perfect information, such as Tic-Tac-Toe. It operates by exploring all possible future moves of both players until it reaches a terminal state, which could be a win, a loss, or a draw. Minimax assumes that both players play optimally, with the maximizer attempting to maximize its score and the minimizer striving to minimize the score.

3. Evaluation Function:

An evaluation function is utilized in Minimax-based algorithms to assess the desirability of a game state for a player. This function assigns a numerical value to each game state, representing its favorability to the

player. A well-designed evaluation function is crucial for the Minimax algorithm to make informed decisions about which moves to select.

4. Alpha-Beta Pruning :

Alpha-Beta pruning is an optimization technique often applied in conjunction with the Minimax algorithm. It aims to reduce the number of nodes evaluated during the search process by pruning branches of the game tree that are deemed irrelevant to the final decision. This optimization significantly enhances the efficiency of the algorithm, particularly in games with large search spaces.

5. Implementation Overview:

In our implementation, we represent the Tic-Tac-Toe game board as a 3x3 grid. We have implemented the Minimax algorithm along with an evaluation function to determine the optimal move for the AI player. Additionally, we have provided an optional implementation of Alpha-Beta pruning to improve the efficiency of our algorithm.

Program Code:

```cpp
// C++ program to find the next optimal move for
// a player
#include<bits/stdc++.h>
using namespace std;

struct Move
{
    int row, col;
};

char player = 'x', opponent = 'o';

// This function returns true if there are moves
// remaining on the board. It returns false if
// there are no moves left to play.
bool isMovesLeft(char board[3][3])
{
    for (int i = 0; i<3; i++)
        for (int j = 0; j<3; j++)
            if (board[i][j]=='_')
                return true;
    return false;
}

// This is the evaluation function as discussed
// in the previous article ( http://goo.gl/sJgv68 )
int evaluate(char b[3][3])
{
    // Checking for Rows for X or O victory.
```

```
    for (int row = 0; row<3; row++)
    {
        if (b[row][0]==b[row][1] &&
            b[row][1]==b[row][2])
        {
            if (b[row][0]==player)
                return +10;
            else if (b[row][0]==opponent)
                return -10;
        }
    }

    // Checking for Columns for X or O victory.
    for (int col = 0; col<3; col++)
    {
        if (b[0][col]==b[1][col] &&
            b[1][col]==b[2][col])
        {
            if (b[0][col]==player)
                return +10;

            else if (b[0][col]==opponent)
                return -10;
        }
    }

    // Checking for Diagonals for X or O victory.
    if (b[0][0]==b[1][1] && b[1][1]==b[2][2])
    {
        if (b[0][0]==player)
            return +10;
        else if (b[0][0]==opponent)
            return -10;
    }

    if (b[0][2]==b[1][1] && b[1][1]==b[2][0])
    {
        if (b[0][2]==player)
            return +10;
        else if (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
int minimax(char board[3][3], int depth, bool isMax)
{
    int 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
    if (isMax)
    {
        int best = -1000;

        // Traverse all cells
        for (int i = 0; i<3; i++)
        {
            for (int j = 0; j<3; j++)
            {
                // 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, !isMax) );

                    // Undo the move
                    board[i][j] = '_';
                }
            }
```

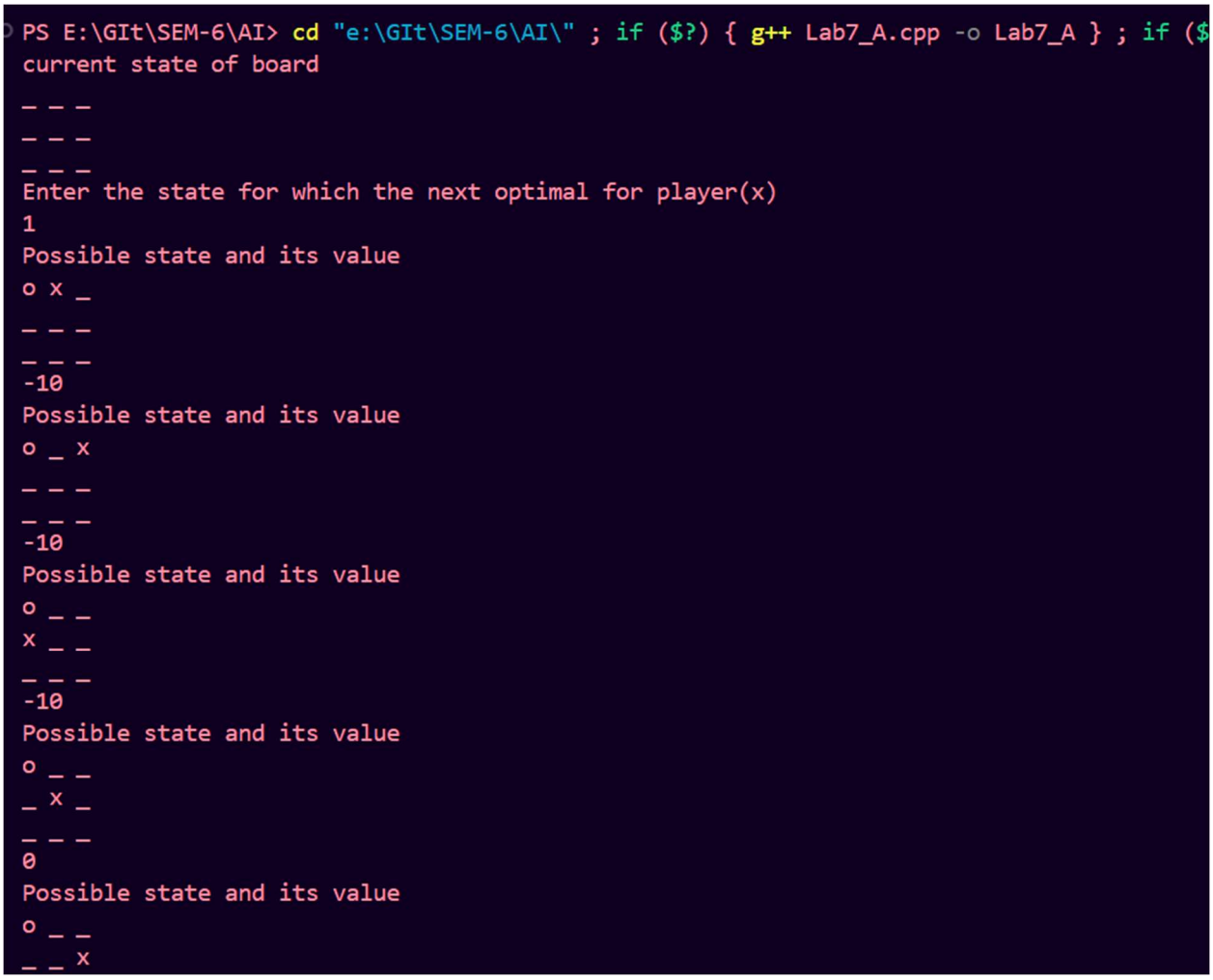
```
        }
        return best;
    }

    // If this minimizer's move
    else
    {
        int best = 1000;

        // Traverse all cells
        for (int i = 0; i<3; i++)
        {
            for (int j = 0; j<3; j++)
            {
                // 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, !isMax));

                    // Undo the move
                    board[i][j] = '_';
                }
            }
        }
        return best;
    }
}


void printBoard(char board[3][3]){
    for(int i=0;i<3;i++){
        for(int j=0;j<3;j++){
            cout<<board[i][j]<<" ";
          }
          cout<<endl;
       }
}

// This will return the best possible move for the player
Move findBestMove(char board[3][3])
{
```

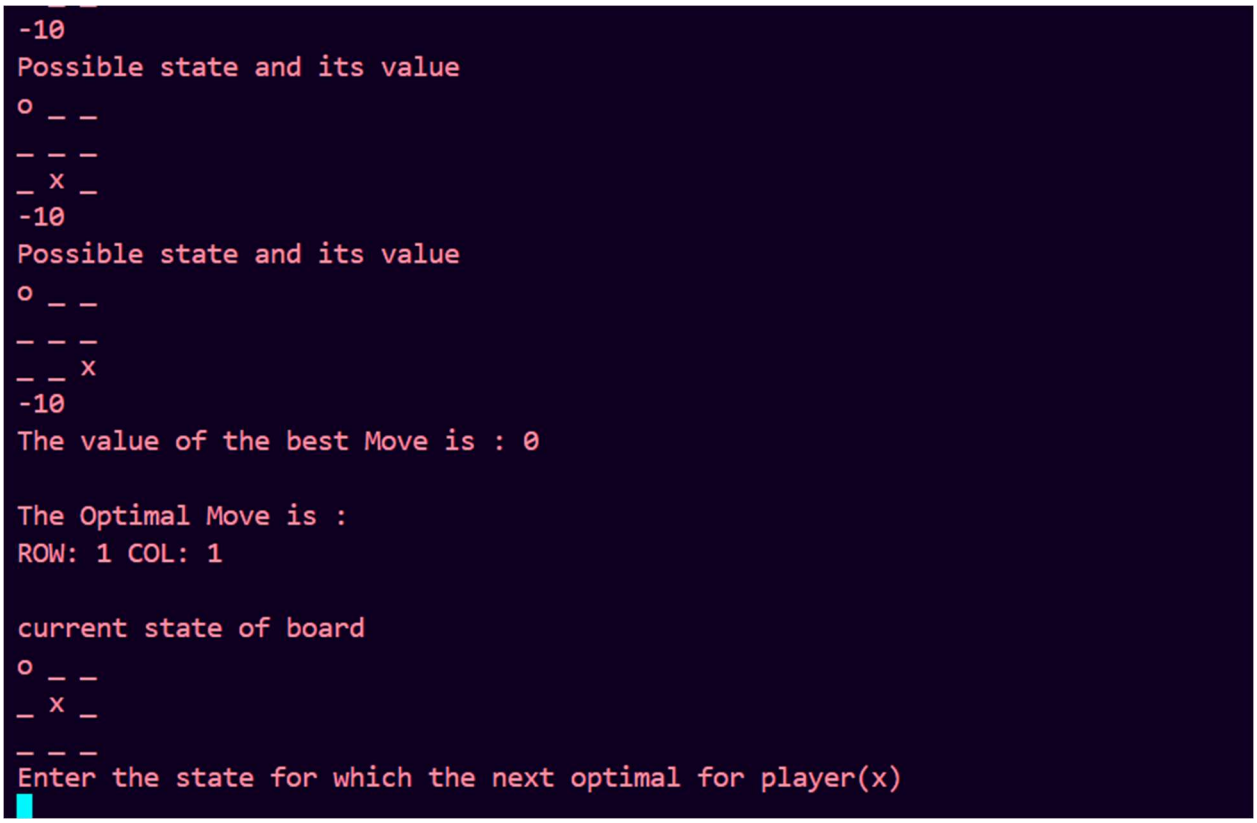
```
    int bestVal = -1000;
    Move bestMove;
    bestMove.row = -1;
    bestMove.col = -1;

    // Traverse all cells, evaluate minimax function for
    // all empty cells. And return the cell with optimal
    // value.
    for (int i = 0; i<3; i++)
    {
        for (int j = 0; j<3; j++)
        {
            // Check if cell is empty
            if (board[i][j]=='_')
            {
                // Make the move
                board[i][j] = player;

                // compute evaluation function for this
                // move.

                int moveVal = minimax(board, 0, false);
                cout<<"Possible state and its value"<<endl;
                printBoard(board);
                cout<<moveVal<<endl;


                // 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.row = i;
                    bestMove.col = j;

                    bestVal = moveVal;
                }
            }

        }
    }

    printf("The value of the best Move is : %d\n\n",
            bestVal);
```

```
    board[bestMove.row][bestMove.col]='x';

    return bestMove;
}

// Driver code
int main()
{

  char board[3][3] =
    {
        { '_', '_', '_' },
        { '_', '_', '_' },
        { '_', '_', '_' }
    };



   while(true){
    cout<<"current state of board"<<endl;
    printBoard(board);

    cout << "Enter the state for which the next optimal for player(x)"<<endl;

    int c;
    cin>>c;

    if(c==1){
        board[0][0]='o';
    }else if(c==2){
        board[0][1]='o';
    }else if(c==3){
        board[0][2]='o';
    }else if(c==4){
        board[1][0]='o';
    }else if(c==5){
        board[1][1]='o';
    }else if(c==6){
        board[1][0]='o';
    }else if(c==7){
        board[2][0]='o';
    }else if(c==8){
        board[2][1]='o';
    }else if(c==9){
        board[2][2]='o';
    }
```

```
    Move bestMove = findBestMove(board);

    printf("The Optimal Move is :\n");
    printf("ROW: %d COL: %d\n\n", bestMove.row,
                                  bestMove.col );

   }
    return 0;
}
```

Output:


PS E:\GIt\SEM-6\AI> cd "e:\GIt\SEM-6\AI\" ; if ($?) { g++ Lab7_A.cpp -o Lab7_A } ; if ($
current state of board
_ _ _
_ _ _
_ _ _
Enter the state for which the next optimal for player(x)
1
Possible state and its value
o x _
_ _ _
_ _ _
-10
Possible state and its value
o _ x
_ _ _
_ _ _
-10
Possible state and its value
o _ _
x _ _
_ _ _
-10
Possible state and its value
o _ _
_ x _
_ _ _
0
Possible state and its value
o _ _
_ _ x



Conclusion:

Through the development of a Tic-Tac-Toe AI player using the Minimax algorithm, we have delved into the realm of artificial intelligence and game theory. This project highlights the significance of strategic decision-making in adversarial environments, showcasing how the Minimax algorithm, coupled with an appropriate evaluation function, enables the AI player to make optimal moves. Additionally, the optional implementation of Alpha-Beta pruning enhances computational efficiency, contributing to faster decision-making. This endeavor underscores the importance of algorithmic optimization and heuristic evaluation in creating intelligent systems capable of competing in strategic games. Moving forward, opportunities for further refinement include integrating advanced algorithms and enhancing the evaluation function. Overall, this project provides valuable insights into AI development, inspiring continued exploration in the field of intelligent systems