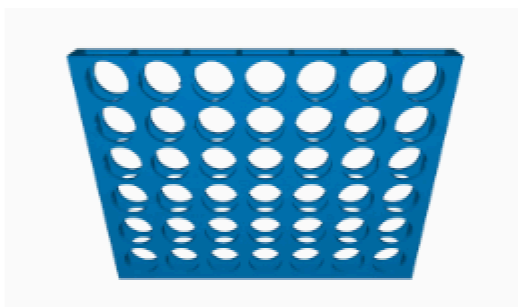
Assignment 04

Name - Sk Fardeen Hossain
Enrollment No - 2021CSB023
Department - Computer Science and Technology

<u>Question 01:-</u>

1. Connect-4 is a strategic two-player game where participants choose a disc colour and take turns dropping their coloured discs into a seven-column, six-row grid.



Victory is achieved by forming a line of four discs horizontally, vertically, or diagonally. Several winning strategies enhance gameplay:

a. Middle Column Placement:

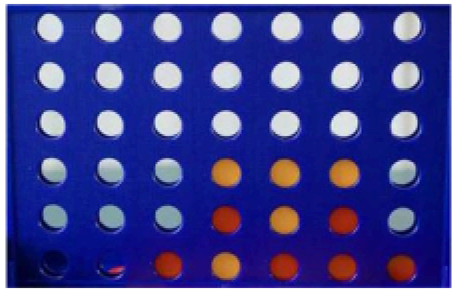
The player initiating the game benefits from placing the first disc in the middle column. This strategic move maximizes the possibilities for vertical, diagonal, and horizontal connections, totalling five potential ways to win.

b. Trapping Opponents:

To prevent losses, players strategically block their opponent's potential winning paths. For instance, placing a disc adjacent to an opponent's three-disc line disrupts their progression and protects the player from falling into traps set by the opponent.

c. "7" Formation:

Employing a "7" trap involves arranging discs to resemble the shape of a 7 on the board. This strategic move, which can be configured in various orientations, provides players with multiple directions to achieve a connect-four, adding versatility to their gameplay.



Connect-4 Implementation using Mini-Max Algorithm:

In this scenario, a user engages in a game against the computer, and the Mini-Max algorithm is employed to generate game states. Mini-Max, a backtracking algorithm widely used in decision-making and game theory, determines the optimal move for a player under the assumption that the opponent also plays optimally. Two players, the maximizer and the minimizer, aim to achieve the highest and lowest scores, respectively. A heuristic function calculates the values associated with each board state, representing the advantage of one player over the other.

Connect-4 Implementation using Alpha-Beta Pruning:

To optimize the Mini-Max algorithm, the Alpha-Beta Pruning technique is applied. Alpha-Beta Pruning involves passing two additional parameters, alpha and beta, to the Mini-Max function, reducing the number of evaluated nodes in the game tree. By introducing these parameters, the algorithm searches more efficiently, reaching greater depths in the game tree. Alpha-Beta Pruning accelerates the search process by eliminating the need to evaluate unnecessary branches when a superior move has been identified, resulting in significant computational time savings.

Idea (MiniMax Algorithm) :

The main idea behind solving interactive game problems, is to first visualise the search space in the form of a ***'Game Tree'***. The game tree consists of all the possible game states that are reachable by following a not necessarily unique sequence of moves. Each of the nodes in the game tree are connected via viable actions (in this case placement of a piece in any one of the columns).

In the connect4 game, the idea of winning comes from the fact that any 4 pieces of a player appear consecutively in the board either horizontally, vertically or diagonally.

Here there are 2 players:- One is the player and another one is the AI agent. The game tree is designed keeping the **AI agent as the maximiser**, that is the score will be positive if the AI agent is close to winning and negative if the player is close to winning.

So each level of the game tree alternates between the maximiser and minimiser, and the structure assumes everyone plays optimally.

Alpha Beta Pruning

The main idea behind alpha beta pruning is that some states are always leading to a given result no matter the sequence of moves, so to save on the memory such states are pruned from the game tree. The alpha value is updated by the maximiser and the beta value is updated by the minimiser. The condition for pruning is "**alpha >= beta**".

Depth Parameter

Now for a given problem like the connect4, the height of the game tree will be huge and storing the states in the memory will be a big problem. So depth is taken as a hyperparameter here, which signifies the max depth of the tree beyond which the nodes will not be expanded and evaluation of the nodes is done whether terminal or non-terminal.

Memoization

A given board can be reachable from the root node from various action of moves, so to prevent the redundant expansion it is better that we cache the states.

Code :

```
#include <bits/stdc++.h>

#define ROWS 6
#define COLS 7
#define PLAYER 1
#define AI 2
#define EMPTY 0

using namespace std;

class Connect4
{
public:
   vector<vector<int>> board;

   Connect4()
   {
       board = vector<vector<int>>(ROWS, vector<int>(COLS, EMPTY));
   }

   void printBoard()
   {
       // Prints the current board configuration
       for (int i = 0; i < ROWS; i++)
       {
           cout << "\n-----------------------------\n|  ";

           for (int j = 0; j < COLS; j++)
           {
               if (board[i][j] == PLAYER)
                   cout << "x" << " | ";
```

```
            else if (board[i][j] == AI)
                cout << "o" << " | ";
            else
                cout << " " << " | ";
        }
    }
    cout << "\n-----------------------------\n";
}

bool isColumnFull(int col)
{
    // Returns true if column is fully occupied
    return board[0][col] != EMPTY;
}

bool placePiece(int col, int player)
{
    // Returns true if the piece can be placed
    for (int i = ROWS - 1; i >= 0; i--)
    {
        if (board[i][col] == EMPTY)
        {
            board[i][col] = player;
            return true;
        }
    }
    return false;
}

bool checkAlignment(int r, int c, int dr, int dc, int player)
{
    /* Returns true if starting from the given (r,c) following
     the trend (dr,dc), we get a length of atleast 4*/
    int cnt = 0;
    for (int i = 0; i < 4; i++)
    {
        int nr = r + i * dr;
        int nc = c + i * dc;
        if (nr >= 0 && nc >= 0 && nr < ROWS && nc < COLS &&
board[nr][nc] == player)
```

```
        {
            cnt++;
        }
        else
        {
            break;
        }
    }
    return cnt == 4;
}

bool checkWin(int player)
{
    // Returns true if the player wins

    for (int i = 0; i < ROWS; i++)
    {
        for (int j = 0; j < COLS; j++)
        {
            if (board[i][j] == player)
            {
                // horizontal
                bool hori_flag = checkAlignment(i, j, 0, 1, player);
                // vertical
                bool vert_flag = checkAlignment(i, j, 1, 0, player);
                // main diag
                bool main_diag = checkAlignment(i, j, 1, 1, player);
                // off diag
                bool off_diag = checkAlignment(i, j, 1, -1, player);

                if (hori_flag || vert_flag || main_diag || off_diag)
                    return true;
            }
        }
    }
    return false;
}

bool isBoardFull()
{
```

```cpp
        // Returns true if board is full

        for (int i = 0; i < COLS; i++)
        {
            if (!isColumnFull(i))
                return false;
        }
        return true;
    }

    string boardHash()
    {
        // Returns a hash-string for the given board config.
        string key = "";
        for (int i = 0; i < ROWS; i++)
        {
            for (int j = 0; j < COLS; j++)
            {
                key += to_string(board[i][j]);
            }
        }
        return key;
    }
};

// Consider AI as the maximiser and player as the minimiser
class MiniMaxSolver
{
public:
    unordered_map<string, int> memo;

    int evaluateBoard(Connect4 &game)
    {
        // Evaluation of terminal nodes in the game tree
        if (game.checkWin(AI))
        {
            return 1000;
        }
        else if (game.checkWin(PLAYER))
        {
```

```
            return -1000;
        }

        // Immediate Threat from player
        for (int i = 0; i < COLS; i++)
        {
            if (!game.isColumnFull(i))
            {
                Connect4 newState = game;
                newState.placePiece(i, PLAYER);
                if (newState.checkWin(PLAYER))
                {
                    return -500;
                }
            }
        }

        // Immediate threat from AI
        for (int i = 0; i < COLS; i++)
        {
            if (!game.isColumnFull(i))
            {
                Connect4 newState = game;
                newState.placePiece(i, AI);
                if (newState.checkWin(AI))
                {
                    return 500;
                }
            }
        }
        return 0;
    }

    int minimax(Connect4 &game, int depth, int alpha, int beta, bool
isMaximizer)
    {
        // EValuates the value of the root node in the game tree keeping AI
as maximiser
        string board_hash = game.boardHash() + (isMaximizer ? "A" : "P");
```

```
        if (memo.find(board_hash) != memo.end())
        {
            return memo[board_hash];
        }

        if (depth == 0 || game.checkWin(AI) || game.checkWin(PLAYER) ||
game.isBoardFull())
        {
            int eval = evaluateBoard(game);
            memo[board_hash] = eval;
            return eval;
        }

        if (isMaximizer)
        {
            int maxEval = INT_MIN;
            for (int i = 0; i < COLS; i++)
            {
                if (!game.isColumnFull(i))
                {
                    Connect4 newState = game;
                    newState.placePiece(i, AI);
                    int eval = minimax(newState, depth - 1, alpha, beta,
false);
                    maxEval = max(maxEval, eval);
                    alpha = max(alpha, eval);
                    if (alpha >= beta)
                    {
                        // prune;
                        break;
                    }
                }
            }
            memo[board_hash] = maxEval;
            return maxEval;
        }
        else
        {
            int minEval = INT_MAX;
            for (int i = 0; i < COLS; i++)
```

```
            {
                if (!game.isColumnFull(i))
                {
                    Connect4 newState = game;
                    newState.placePiece(i, PLAYER);
                    int eval = minimax(newState, depth - 1, alpha, beta,
true);
                    minEval = min(minEval, eval);
                    beta = min(beta, eval);
                    if (alpha >= beta)
                    {
                        // prune
                        break;
                    }
                }
            }
            memo[board_hash] = minEval;
            return minEval;
        }
    }

    int optimalAction(Connect4 &game, int depth)
    {
        // Returns optimal Action from a given board state
        int optimalAction = -1;
        int optimalValue = INT_MIN;
        for (int i = 0; i < COLS; i++)
        {
            if (!game.isColumnFull(i))
            {
                Connect4 newState = game;
                newState.placePiece(i, AI);
                int moveVal = minimax(newState, depth - 1, INT_MIN,
INT_MAX, false);
                if (moveVal > optimalValue)
                {
                    optimalValue = moveVal;
                    optimalAction = i;
                }
            }
```

```cpp
        }
        return optimalAction;
    }
};

int main()
{
    Connect4 game;
    MiniMaxSolver ai_agent;
    int depth; // Max depth of the game tree
    cout << "Your symbol -> x\n";
    cout << "AI's symbol -> o\n";
    cout << "Enter the depth of the Game Tree: ";
    cin >> depth;
    while (true)
    {
        game.printBoard();

        int playerCol;
        cout << "Enter your move (1-7): ";
        cin >> playerCol;
        playerCol--;
        if (playerCol < 0 || playerCol >= COLS ||
game.isColumnFull(playerCol))
        {
            cout << "Invalid mode, try again\n";
            continue;
        }

        game.placePiece(playerCol, PLAYER);
        if (game.checkWin(PLAYER))
        {
            game.printBoard();
            cout << "You win!\n";
            return 0;
        }
        if (game.isBoardFull())
        {
            game.printBoard();
            cout << "It's a tie\n";
```

```
            return 0;
        }

        int aiCol = ai_agent.optimalAction(game, depth);
        game.placePiece(aiCol, AI);
        if (game.checkWin(AI))
        {
            game.printBoard();
            cout << "AI wins!\n";
            return 0;
        }
        if (game.isBoardFull())
        {
            game.printBoard();
            cout << "It's a tie\n";
            return 0;
        }
    }
    return 0;
}
```

Input :

It is an alternate sequence of moves between the player (user) and the AI agent.

Output :

```
Your symbol -> x
AI's symbol -> o
Enter the depth of the Game Tree: 3

-----------------------------
|   |   |   |   |   |   |   |
-----------------------------
|   |   |   |   |   |   |   |
-----------------------------
|   |   |   |   |   |   |   |
-----------------------------
|   |   |   |   |   |   |   |
-----------------------------
|   |   |   |   |   |   |   |
-----------------------------
|   |   |   |   |   |   |   |
-----------------------------
Enter your move (1-7): 4

-----------------------------
|   |   |   |   |   |   |   |
-----------------------------
|   |   |   |   |   |   |   |
-----------------------------
|   |   |   |   |   |   |   |
-----------------------------
|   |   |   |   |   |   |   |
-----------------------------
|   |   |   |   |   |   |   |
-----------------------------
| o |   |   | x |   |   |   |
-----------------------------
Enter your move (1-7): 4
```

```
-----------------------------
|   |   |   |   |   |   |   |
-----------------------------
|   |   |   |   |   |   |   |
-----------------------------
|   |   |   |   |   |   |   |
-----------------------------
|   |   |   |   |   |   |   |
-----------------------------
| o |   |   | x |   |   |   |
-----------------------------
| o |   |   | x |   |   |   |
-----------------------------
Enter your move (1-7): 4

-----------------------------
|   |   |   |   |   |   |   |
-----------------------------
|   |   |   |   |   |   |   |
-----------------------------
|   |   |   | o |   |   |   |
-----------------------------
|   |   |   | x |   |   |   |
-----------------------------
| o |   |   | x |   |   |   |
-----------------------------
| o |   |   | x |   |   |   |
-----------------------------
Enter your move (1-7): 4
```

```
-----------------------------
|   |   |   |   |   |   |   |
-----------------------------
|   |   |   | x |   |   |   |
-----------------------------
|   |   |   | o |   |   |   |
-----------------------------
| o |   |   | x |   |   |   |
-----------------------------
| o |   |   | x |   |   |   |
-----------------------------
| o |   |   | x |   |   |   |
-----------------------------
Enter your move (1-7): 4

-----------------------------
|   |   |   | x |   |   |   |
-----------------------------
|   |   |   | x |   |   |   |
-----------------------------
| o |   |   | o |   |   |   |
-----------------------------
| o |   |   | x |   |   |   |
-----------------------------
| o |   |   | x |   |   |   |
-----------------------------
| o |   |   | x |   |   |   |
-----------------------------
AI wins!
```