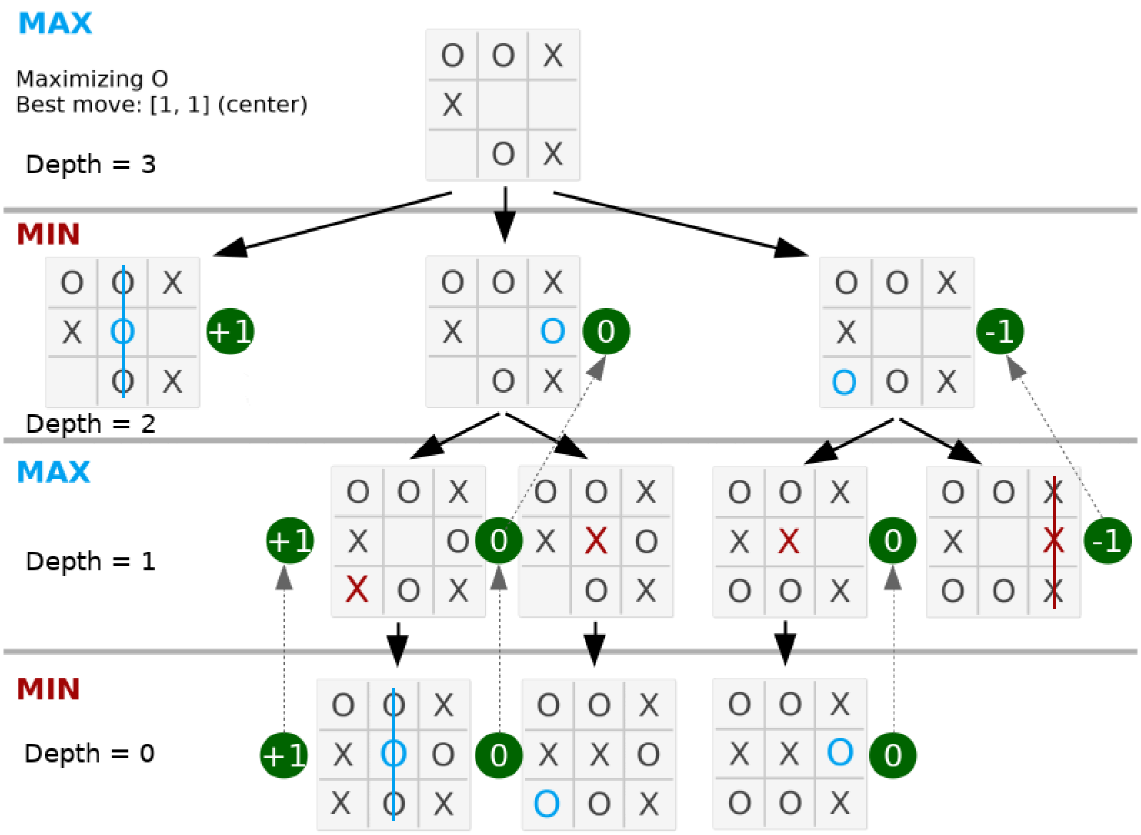
Bharatiya Vidya Bhavan's

Sardar Patel Institute of Technology

Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai-400058-India
(Autonomous College Affiliated to University of Mumbai)

SUB: AIML
Branch: COMPS B

Name :	**Chetan Deepak Patil**
UID :	**2023301012**
Exp no :	**04**
Aim :	**Implement the problem using the Informed searching technique min-max algorithm . Analyze the algorithm with respect to Completeness, Optimality, time and space Complexity** **a) Tic Tac Toe** **b) Implement Alpha Beta Pruning Algorithm on the same Game and put comparison**
Thesis :	**Completeness, Optimality, Time, and Space Complexity:** ● **Completeness**: The Minimax algorithm is complete. It guarantees that a solution (win, lose, or draw) will be found if there is one because it explores all possible moves. ● **Optimality**: The Minimax algorithm is optimal when both players play perfectly. It always chooses the best possible move assuming the opponent is also playing optimally. ● **Time Complexity**: The time complexity of the Minimax algorithm is **O(b^d)**, where: ○ **b** = branching factor (number of possible moves per turn, up to 9 for Tic Tac Toe) ○ **d** = depth of the game tree (up to 9 for Tic Tac Toe) ● For Tic Tac Toe, the maximum time complexity is O(9!) or approximately 362,880 in the worst case, but pruning techniques like alpha-beta can reduce this. ● **Space Complexity**: The space complexity of the Minimax algorithm is **O(d)**, where: ○ **d** = maximum depth of the game tree (which is 9 for Tic Tac Toe) ● The space complexity depends on the depth of the recursion stack, which, in the case of Tic Tac Toe, is limited to 9 levels.



Analysis of Alpha-Beta Pruning:

- **Completeness**:
  - Alpha-Beta Pruning is also **complete.** It prunes irrelevant branches but still guarantees finding the optimal solution.
- **Optimality**:
  - It is **optimal** for the same reason as the Minimax algorithm; it ensures that the maximizing player (AI) gets the best result assuming optimal play.
- **Time Complexity**:

  - **O(b^d/2)** in the best case, which is much better than regular Minimax.
  - In Tic-Tac-Toe, this can be up to half the nodes evaluated, significantly reducing time.
- **Space Complexity**:
  - Same as Minimax: **O(b*d)**, as the space complexity depends on the recursive depth and the game tree.

---

3. Comparison Between Minimax and Alpha-Beta Pruning

Aspect	Minimax	Alpha-Beta Pruning
Completeness	Yes	Yes
Optimality	Yes	Yes
Time Complexity	O(b^d)	O(b^(d/2)) (best case)
Space Complexity	O(b*d)	O(b*d)
Performance	Slower	Faster due to pruning

Alpha-Beta Pruning is a clear improvement over Minimax in terms of time complexity, especially when applied to larger game trees. However, in smaller games like Tic-Tac-Toe, the difference is minimal, but Alpha-Beta Pruning can still make the algorithm more efficient.

Code :

```
#include <iostream>
#include <vector>
#include <limits.h>


using namespace std;


#define PLAYER 'X'  // Maximizing player
#define OPPONENT 'O'  // Minimizing player


// Function to print the board
void printBoard(char board[3][3]) {
   for (int row = 0; row < 3; row++) {
       for (int col = 0; col < 3; col++) {
           cout << board[row][col] << " ";
       }
       cout << endl;
   }
```

```
}



// Function to check if there are moves left on the board
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;
}



// Function to evaluate the board state
int evaluate(char board[3][3]) {
    // Check rows for victory
    for (int row = 0; row < 3; row++) {
        if (board[row][0] == board[row][1] && board[row][1] == board[row][2]) {
            if (board[row][0] == PLAYER)
                return +10;
            else if (board[row][0] == OPPONENT)
                return -10;
        }
    }


    // Check columns for victory
    for (int col = 0; col < 3; col++) {
        if (board[0][col] == board[1][col] && board[1][col] ==
board[2][col]) {
            if (board[0][col] == PLAYER)
                return +10;
            else if (board[0][col] == OPPONENT)
                return -10;
        }
    }


    // Check diagonals for victory
    if (board[0][0] == board[1][1] && board[1][1] == board[2][2]) {
        if (board[0][0] == PLAYER)
```

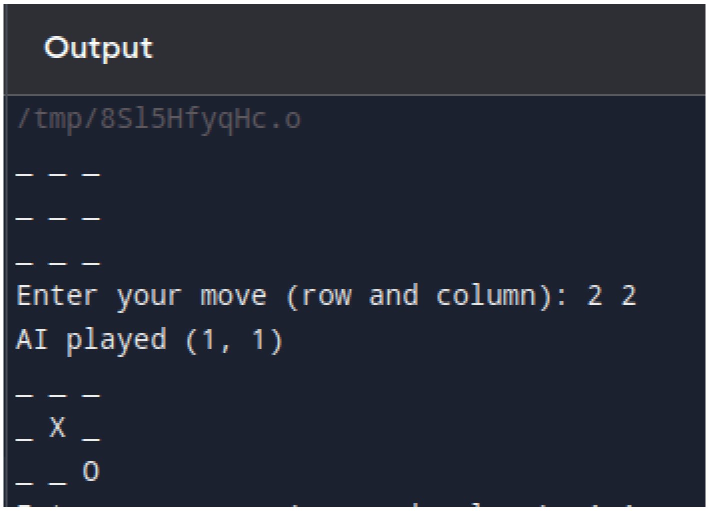
```
            return +10;
        else if (board[0][0] == OPPONENT)
            return -10;
    }


    if (board[0][2] == board[1][1] && board[1][1] == board[2][0]) {
        if (board[0][2] == PLAYER)
            return +10;
        else if (board[0][2] == OPPONENT)
            return -10;
    }


    // No winner: return 0
    return 0;
}


// Minimax algorithm
int minimax(char board[3][3], int depth, bool isMax) {
    int score = evaluate(board);

    // If Maximizer or Minimizer has won, return the score
    if (score == 10)
        return score - depth; // Subtract depth to prioritize shorter win
    if (score == -10)
        return score + depth; // Add depth to prioritize shorter loss

    // If no moves are left, it's a draw
    if (!isMovesLeft(board))
        return 0;

    // Maximizing player's move
    if (isMax) {
        int best = INT_MIN;
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                if (board[i][j] == '_') {
                    board[i][j] = PLAYER;
                    best = max(best, minimax(board, depth + 1, false));
                    board[i][j] = '_';
                }
            }
        }
        return best;
    }
    // Minimizing player's move
    else {
        int best = INT_MAX;
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                if (board[i][j] == '_') {
                    board[i][j] = OPPONENT;
                    best = min(best, minimax(board, depth + 1, true));
                    board[i][j] = '_';
                }
            }
        }

        return best;

    }

}



// Function to find the best move for the player

pair<int, int> findBestMove(char board[3][3]) {

    int bestVal = INT_MIN;

    pair<int, int> bestMove = {-1, -1};



    for (int i = 0; i < 3; i++) {

        for (int j = 0; j < 3; j++) {

            if (board[i][j] == '_') {

                board[i][j] = PLAYER;

                int moveVal = minimax(board, 0, false);

                board[i][j] = '_';



                if (moveVal > bestVal) {
                    bestMove = {i, j};
                    bestVal = moveVal;
                }
            }
        }
    }


    return bestMove;
}



int main() {
    char board[3][3] = {
        {'_', '_', '_'},
        {'_', '_', '_'},
        {'_', '_', '_'}
    };


    while (isMovesLeft(board)) {
        int row, col;
        printBoard(board);

        // Ask the user (opponent) for their move
        cout << "Enter your move (row and column): ";
        cin >> row >> col;


        // Validate the user input
        if (row < 0 || row > 2 || col < 0 || col > 2 || board[row][col] != '_') {
            cout << "Invalid move. Please try again." << endl;
            continue;
        }


        // Make the opponent's move
        board[row][col] = OPPONENT;


        // Find the best move for the AI player
        pair<int, int> bestMove = findBestMove(board);



        // Make the best move for the AI player
        if (isMovesLeft(board) && bestMove.first != -1 && bestMove.second != -1) {
            board[bestMove.first][bestMove.second] = PLAYER;
            cout << "AI played (" << bestMove.first << ", " << bestMove.second << ")\n";
        }



        // Check if the game has a winner
        int result = evaluate(board);
        if (result == 10) {
            cout << "AI wins!" << endl;
            break;
        } else if (result == -10) {
            cout << "You win!" << endl;
            break;
        } else if (!isMovesLeft(board)) {
            cout << "It's a draw!" << endl;
            break;

        }

    }




    printBoard(board);

    return 0;

}
```

Code for Alpha Beta pruning :-

```
#include <iostream>
#include <limits.h>

using namespace std;

#define PLAYER 'X'
#define OPPONENT 'O'

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

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;
}

int evaluate(char board[3][3]) {
    for (int row = 0; row < 3; row++) {
        if (board[row][0] == board[row][1] && board[row][1] ==
board[row][2]) {
            if (board[row][0] == PLAYER)
                return +10;
            else if (board[row][0] == OPPONENT)
                return -10;
        }
    }

    for (int col = 0; col < 3; col++) {
        if (board[0][col] == board[1][col] && board[1][col] ==
board[2][col]) {
            if (board[0][col] == PLAYER)
                return +10;
            else if (board[0][col] == OPPONENT)
                return -10;
        }
    }

    if (board[0][0] == board[1][1] && board[1][1] == board[2][2]) {
        if (board[0][0] == PLAYER)
            return +10;
        else if (board[0][0] == OPPONENT)
            return -10;
    }

    if (board[0][2] == board[1][1] && board[1][1] == board[2][0]) {
        if (board[0][2] == PLAYER)
            return +10;
        else if (board[0][2] == OPPONENT)
```

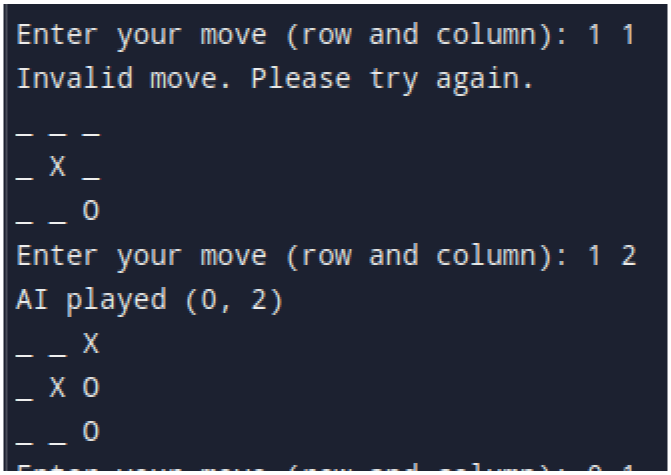
```
            return -10;
    }

    return 0;
}

int alphaBeta(char board[3][3], int depth, bool isMax, int alpha, 
int beta) {
    int score = evaluate(board);

    if (score == 10)
        return score - depth;
    if (score == -10)
        return score + depth;

    if (!isMovesLeft(board))
        return 0;

    if (isMax) {
        int best = INT_MIN;
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                if (board[i][j] == '_') {
                    board[i][j] = PLAYER;
                    best = max(best, alphaBeta(board, depth + 1, 
false, alpha, beta));
                    board[i][j] = '_';
                    alpha = max(alpha, best);
                    if (beta <= alpha)
                        break;
                }
            }
        }
        return best;
    } else {
        int best = INT_MAX;
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                if (board[i][j] == '_') {
                    board[i][j] = OPPONENT;
                    best = min(best, alphaBeta(board, depth + 1,
true, alpha, beta));
                    board[i][j] = '_';
                    beta = min(beta, best);
                    if (beta <= alpha)
                        break;
                }
            }
        }
        return best;
    }
}

pair<int, int> findBestMoveAlphaBeta(char board[3][3]) {
    int bestVal = INT_MIN;
    pair<int, int> bestMove = {-1, -1};

    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (board[i][j] == '_') {
                board[i][j] = PLAYER;
                int moveVal = alphaBeta(board, 0, false, INT_MIN,
INT_MAX);
                board[i][j] = '_';

                if (moveVal > bestVal) {
                    bestMove = {i, j};
                    bestVal = moveVal;
                }
            }
        }
    }

    return bestMove;
}

int main() {
    char board[3][3] = {
        {'_', '_', '_'},
        {'_', '_', '_'},
        {'_', '_', '_'}
    };

    while (isMovesLeft(board)) {
        int row, col;
        printBoard(board);

        cout << "Enter your move (row and column): ";
        cin >> row >> col;

        if (row < 0 || row > 2 || col < 0 || col > 2 ||
board[row][col] != '_') {
            cout << "Invalid move. Please try again." << endl;
            continue;
        }

        board[row][col] = OPPONENT;

        pair<int, int> bestMove = findBestMoveAlphaBeta(board);

        if (isMovesLeft(board) && bestMove.first != -1 &&
bestMove.second != -1) {
            board[bestMove.first][bestMove.second] = PLAYER;
            cout << "AI played (" << bestMove.first << ", " <<
bestMove.second << ")\n";
        }

        int result = evaluate(board);
        if (result == 10) {
            cout << "AI wins!" << endl;
            break;
        } else if (result == -10) {
            cout << "You win!" << endl;
            break;
        } else if (!isMovesLeft(board)) {
            cout << "It's a draw!" << endl;
            break;
        }
    }

    printBoard(board);
    return 0;
}
```

Output :

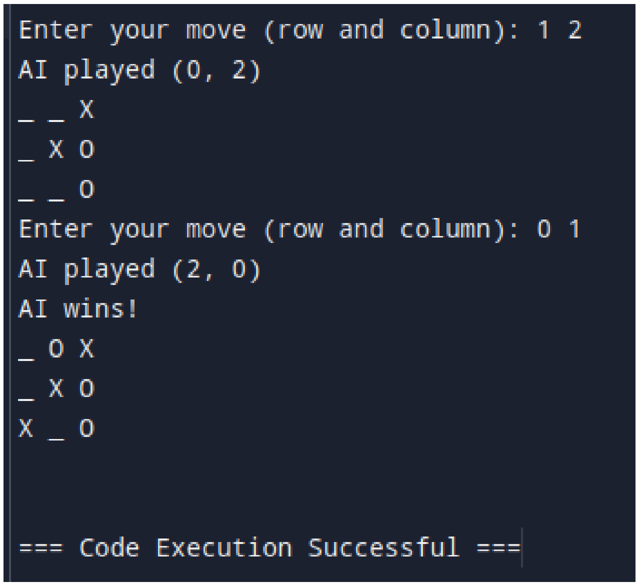


Output

```
/tmp/8Sl5HfyqHc.o
_ _ _
_ _ _
_ _ _
Enter your move (row and column): 2 2
AI played (1, 1)
_ _ _
_ X _
_ _ O
```

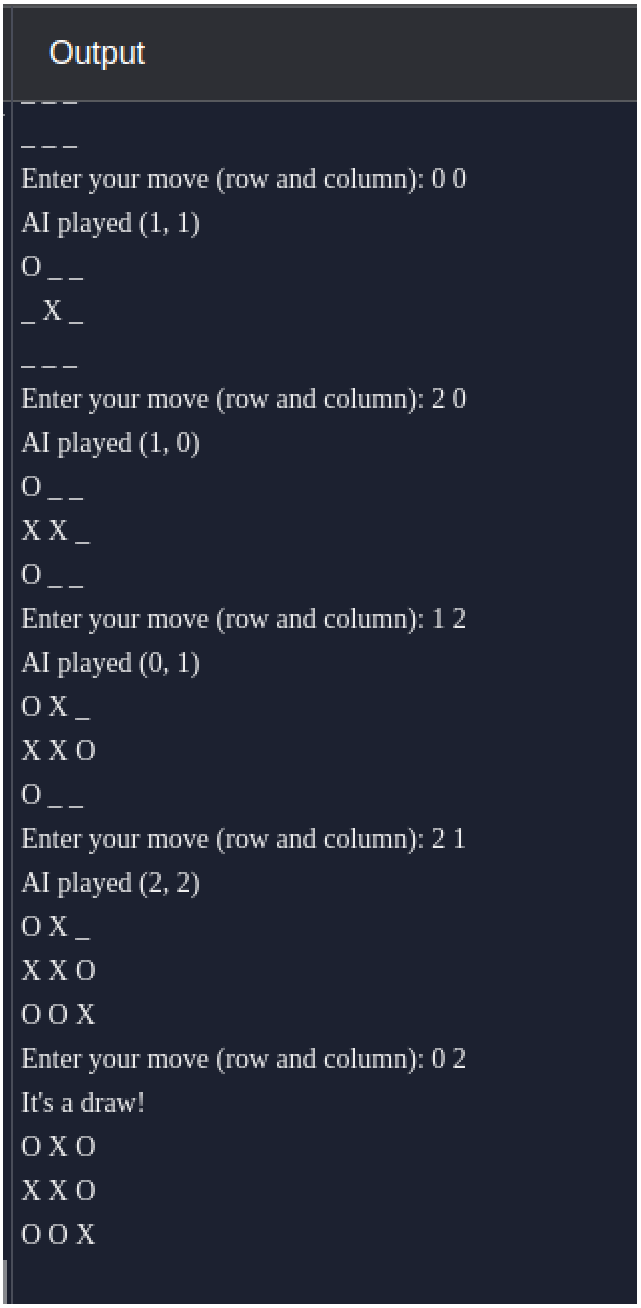


```
Enter your move (row and column): 1 1
Invalid move. Please try again.
_ _ _
_ X _
_ _ O
Enter your move (row and column): 1 2
AI played (0, 2)
_ _ X
_ X O
_ _ O
```



```
Enter your move (row and column): 1 2
AI played (0, 2)
_ _ X
_ X O
_ _ O
Enter your move (row and column): 0 1
AI played (2, 0)
AI wins!
_ O X
_ X O
X _ O


=== Code Execution Successful ===
```

Using Alpha Beta pruning :-



```
_ _ _
_ _ _
Enter your move (row and column): 0 0
AI played (1, 1)
O _ _
_ X _
_ _ _
Enter your move (row and column): 2 0
AI played (1, 0)
O _ _
X X _
O _ _
Enter your move (row and column): 1 2
AI played (0, 1)
O X _
X X O
O _ _
Enter your move (row and column): 2 1
AI played (2, 2)
O X _
X X O
O O X
Enter your move (row and column): 0 2
It's a draw!
O X O
X X O
O O X
```

Conclusion :	The Minimax algorithm is a powerful method for playing Tic Tac Toe optimally. It is both complete and optimal under the assumption of perfect play. However, its time complexity can be high for more complex games, but for Tic Tac Toe, it is manageable due to the relatively small state space. Additionally, techniques like alpha-beta pruning can significantly improve its performance.

	Alpha-Beta Pruning optimizes the Minimax algorithm by reducing the number of nodes evaluated in the game tree, significantly improving efficiency without compromising the outcome. By pruning irrelevant branches, it allows faster decision-making, especially in games with larger search spaces.