Name: Satyam Jaiswal **UID:** 2021600028 **Batch**: B **Date:** 26/09/2023

EXP 5: Hill Climbing

Problem: Implement Random restart hill climbing to solve N queens problem.

1-show no of restarts required to solve the problem

2- show intermediate heuristics at every stage

3- show no of steps required to get the final solution

Program:

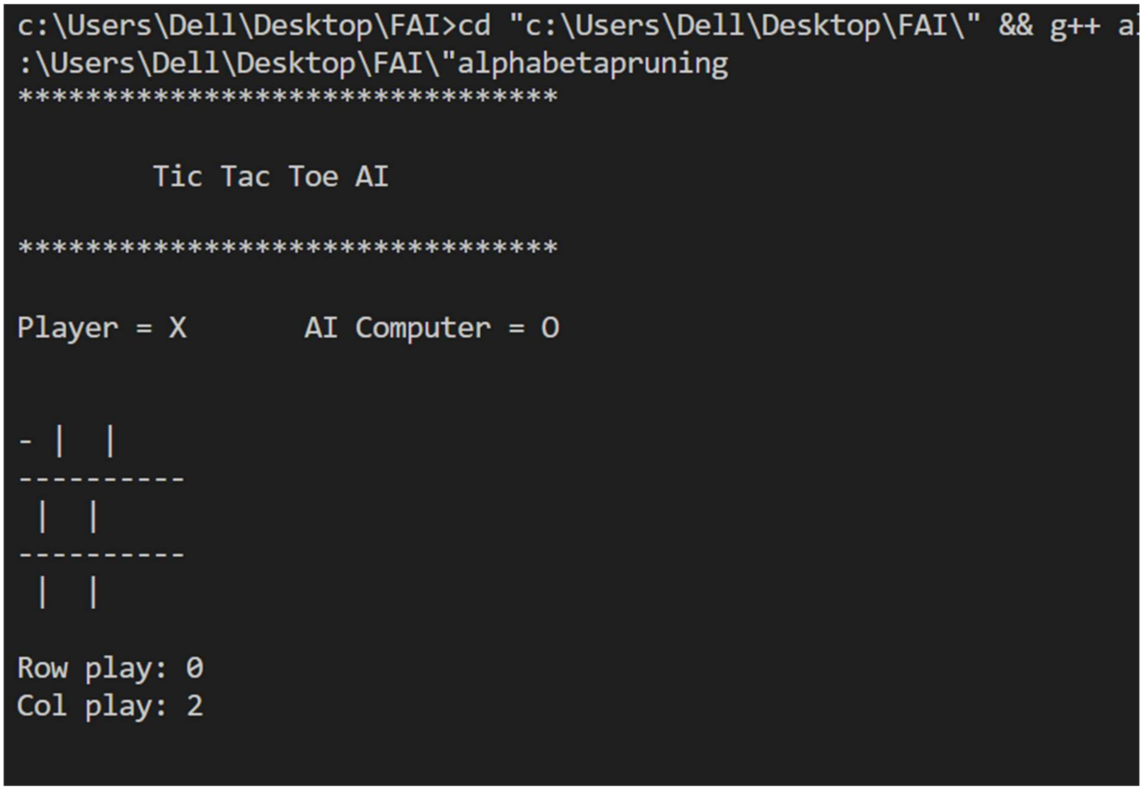
```
#include <bits/stdc++.h>
using namespace std;
#define WIN 1000
#define DRAW 0
#define LOSS -1000
#define AI_MARKER 'O'
#define PLAYER_MARKER 'X'
#define EMPTY_SPACE '-'
#define START_DEPTH 0

// Function prototype
pair<int, pair<int, int>> minimax_optimization(char board[3][3], char marker, int depth, int alpha, int beta);

void print_game_state(int state)
{
   if (WIN == state) { cout << "WIN" << endl; }
   else if (DRAW == state) { cout << "DRAW" << endl; }
   else if (LOSS == state) { cout << "LOSS" << endl; }
}

vector<vector<pair<int, int>>> winning_states
```

```
{
    { make_pair(0, 0), make_pair(0, 1), make_pair(0, 2) },
    { make_pair(1, 0), make_pair(1, 1), make_pair(1, 2) },
    { make_pair(2, 0), make_pair(2, 1), make_pair(2, 2) },

    { make_pair(0, 0), make_pair(1, 0), make_pair(2, 0) },
    { make_pair(0, 1), make_pair(1, 1), make_pair(2, 1) },
    { make_pair(0, 2), make_pair(1, 2), make_pair(2, 2) },

    { make_pair(0, 0), make_pair(1, 1), make_pair(2, 2) },
    { make_pair(2, 0), make_pair(1, 1), make_pair(0, 2) }
};

void print_board(char board[3][3])
{
    cout << endl;
    cout << board[0][0] << " | " << board[0][1] << " | " << board[0][2] << endl;
    cout << "----------" << endl;
    cout << board[1][0] << " | " << board[1][1] << " | " << board[1][2] << endl;
    cout << "----------" << endl;
    cout << board[2][0] << " | " << board[2][1] << " | " << board[2][2] << endl << endl;
}

vector<pair<int, int>> get_legal_moves(char board[3][3])
{
    vector<pair<int, int>> legal_moves;
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            if (board[i][j] != AI_MARKER && board[i][j] != PLAYER_MARKER)
            {
                legal_moves.push_back(make_pair(i, j));
            }
        }
    }

    return legal_moves;
}

bool position_occupied(char board[3][3], pair<int, int> pos)
{
    vector<pair<int, int>> legal_moves = get_legal_moves(board);

    for (int i = 0; i < legal_moves.size(); i++)
    {
        if (pos.first == legal_moves[i].first && pos.second == legal_moves[i].second)
        {
            return false;
        }
    }

    return true;
}

vector<pair<int, int>> get_occupied_positions(char board[3][3], char marker)
{
    vector<pair<int, int>> occupied_positions;

    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            if (marker == board[i][j])
            {
                occupied_positions.push_back(make_pair(i, j));
            }
        }
    }

    return occupied_positions;
}

bool board_is_full(char board[3][3])
{
    vector<pair<int, int>> legal_moves = get_legal_moves(board);

    if (0 == legal_moves.size())
    {
        return true;
    }
    else
    {
        return false;
    }
}

bool game_is_won(vector<pair<int, int>> occupied_positions)
{
    bool game_won;
    for (int i = 0; i < winning_states.size(); i++)
    {
        game_won = true;
        vector<pair<int, int>> curr_win_state = winning_states[i];
        for (int j = 0; j < 3; j++)
        {
            if (!(find(begin(occupied_positions), end(occupied_positions), curr_win_state[j]) != end(occupied_positions)))
            {
                game_won = false;
                break;
            }
        }

        if (game_won)
        {
            break;
        }
    }
    return game_won;
}

char get_opponent_marker(char marker)
{
    char opponent_marker;
    if (marker == PLAYER_MARKER)
    {
        opponent_marker = AI_MARKER;
    }
    else
    {
        opponent_marker = PLAYER_MARKER;
    }

    return opponent_marker;
}

int get_board_state(char board[3][3], char marker)
{
    char opponent_marker = get_opponent_marker(marker);

    vector<pair<int, int>> occupied_positions = get_occupied_positions(board, marker);

    bool is_won = game_is_won(occupied_positions);

    if (is_won)
    {
        return WIN;
    }

    occupied_positions = get_occupied_positions(board, opponent_marker);
    bool is_lost = game_is_won(occupied_positions);

    if (is_lost)
    {
        return LOSS;
    }

    bool is_full = board_is_full(board);
    if (is_full)
    {
        return DRAW;
    }

    return DRAW;
}

pair<int, pair<int, int>> minimax_optimization(char board[3][3], char marker, int depth, int
alpha, int beta)
{
    pair<int, int> best_move = make_pair(-1, -1);
    int best_score = (marker == AI_MARKER) ? LOSS : WIN;
    if (board_is_full(board) || DRAW != get_board_state(board, AI_MARKER))
    {
        best_score = get_board_state(board, AI_MARKER);
        return make_pair(best_score, best_move);
    }

    vector<pair<int, int>> legal_moves = get_legal_moves(board);

    for (int i = 0; i < legal_moves.size(); i++)
    {
        pair<int, int> curr_move = legal_moves[i];
        board[curr_move.first][curr_move.second] = marker;
        if (marker == AI_MARKER)
        {
            int score = minimax_optimization(board, PLAYER_MARKER, depth + 1, alpha,
beta).first;
            if (best_score < score)
            {
                best_score = score + depth * 10;
                best_move = curr_move;
                alpha = max(alpha, best_score);
                board[curr_move.first][curr_move.second] = EMPTY_SPACE;
                if (beta <= alpha)
                {
                    break;
                }
            }
        }
        else
        {
            int score = minimax_optimization(board, AI_MARKER, depth + 1, alpha, beta).first;
            if (best_score > score)
            {
                best_score = score + depth * 10;
                best_move = curr_move;
                beta = min(beta, best_score);
                board[curr_move.first][curr_move.second] = EMPTY_SPACE;
                if (beta <= alpha)
                {
                    break;
                }
            }
        }
        board[curr_move.first][curr_move.second] = EMPTY_SPACE;
    }

    return make_pair(best_score, best_move);
}
```

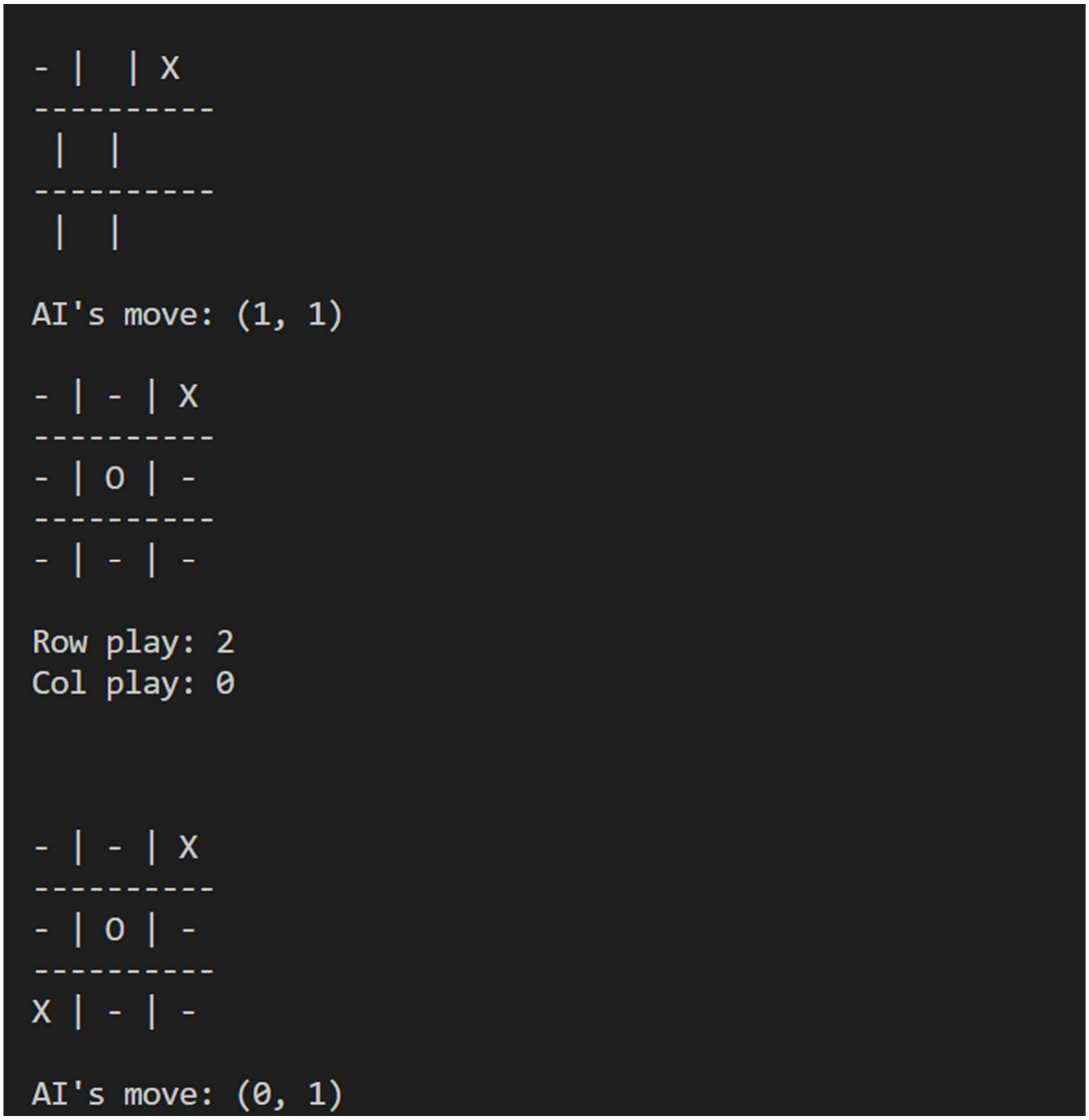
```
bool game_is_done(char board[3][3])
{
    if (board_is_full(board))
    {
        return true;
    }
    if (DRAW != get_board_state(board, AI_MARKER))
    {
        return true;
    }

    return false;
}

// Function to get suggested moves for the human player
vector<pair<int, int>> get_suggested_moves(char board[3][3], char marker)
{
    vector<pair<int, int>> suggested_moves;

    // Create a temporary board to simulate moves and evaluate them
    char temp_board[3][3];
    memcpy(temp_board, board, sizeof(temp_board));

    vector<pair<int, int>> legal_moves = get_legal_moves(temp_board);

    for (int i = 0; i < legal_moves.size(); i++)
    {
        pair<int, int> curr_move = legal_moves[i];
        temp_board[curr_move.first][curr_move.second] = marker;
        int score = minimax_optimization(temp_board, get_opponent_marker(marker),
START_DEPTH, LOSS, WIN).first;
```

```
        if (score == WIN)
        {
            suggested_moves.push_back(curr_move);
        }
        temp_board[curr_move.first][curr_move.second] = EMPTY_SPACE;
    }

    return suggested_moves;
}

int main()
{
    char board[3][3] = { EMPTY_SPACE };
    cout << "********************************\n\n\tTic Tac Toe
AI\n\n********************************" << endl << endl;
    cout << "Player = X\t AI Computer = O" << endl << endl;
    print_board(board);

    while (!game_is_done(board))
    {
        // Calculate and display the suggested moves for the human player
        if (!board_is_full(board) && DRAW != get_board_state(board, AI_MARKER))
        {
            cout << "AI suggests moves for you: ";
            vector<pair<int, int>> suggested_moves = get_suggested_moves(board,
PLAYER_MARKER);
            for (int i = 0; i < suggested_moves.size(); i++)
            {
                cout << "(" << suggested_moves[i].first << ", " << suggested_moves[i].second <<
") ";
            }
            cout << endl;
        }

        int row, col;
        cout << "Row play: ";
        cin >> row;
        cout << "Col play: ";
        cin >> col;
        cout << endl << endl;

        if (position_occupied(board, make_pair(row, col)))
        {
            cout << "The position (" << row << ", " << col << ") is occupied. Try another one..." << endl;
            continue;
        }
        else
        {
            board[row][col] = PLAYER_MARKER;
        }

        print_board(board);
        if (!game_is_done(board))
        {
            cout << "AI's move: ";
            pair<int, pair<int, int>> ai_move = minimax_optimization(board, AI_MARKER, START_DEPTH, LOSS, WIN);
            board[ai_move.second.first][ai_move.second.second] = AI_MARKER;
            cout << "(" << ai_move.second.first << ", " << ai_move.second.second << ")" << endl;
            print_board(board);
        }
    }

    cout << "********** GAME OVER **********" << endl << endl;
    int player_state = get_board_state(board, PLAYER_MARKER);
    cout << "PLAYER "; print_game_state(player_state);
    return 0;
}
```

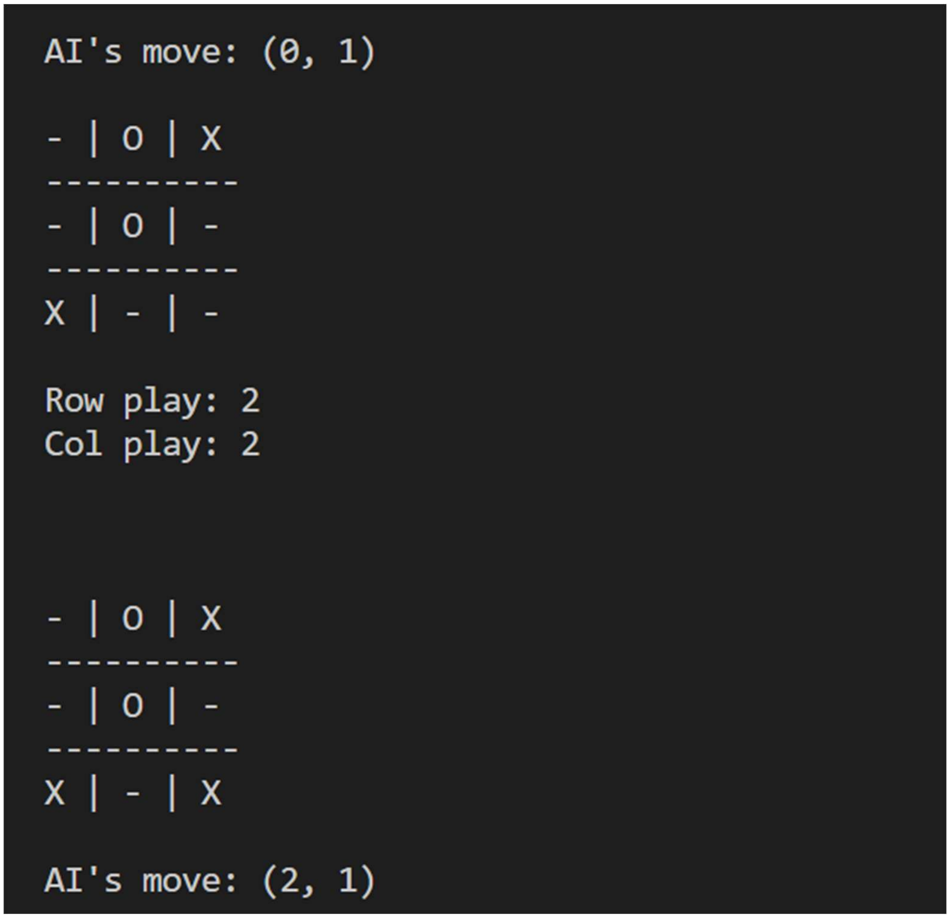
Output:



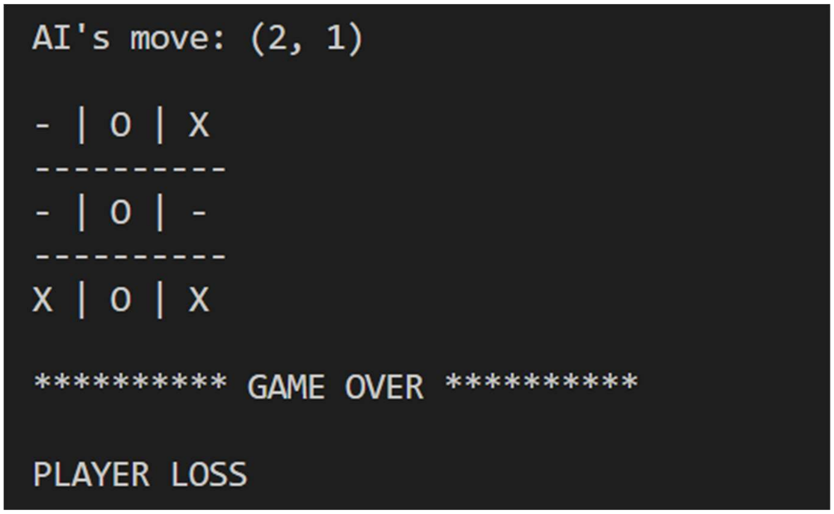
```
c:\Users\Dell\Desktop\FAI>cd "c:\Users\Dell\Desktop\FAI\" && g++ a
:\Users\Dell\Desktop\FAI\"alphabetapruning
********************************

        Tic Tac Toe AI

********************************

Player = X       AI Computer = O


- |  |
----------
 |  |
----------
 |  |

Row play: 0
Col play: 2
```



```
- |  | X
----------
 |  |
----------
 |  |

AI's move: (1, 1)

- | - | X
----------
- | O | -
----------
- | - | -

Row play: 2
Col play: 0



- | - | X
----------
- | O | -
----------
X | - | -

AI's move: (0, 1)
```



```
AI's move: (0, 1)

- | O | X
----------
- | O | -
----------
X | - | -

Row play: 2
Col play: 2



- | O | X
----------
- | O | -
----------
X | - | X

AI's move: (2, 1)
```



```
AI's move: (2, 1)

- | O | X
----------
- | O | -
----------
X | O | X

********** GAME OVER **********

PLAYER LOSS
```