**Name – Armaan Asif Shaikh**

**PRN- 22311412**

**Roll No - 382041**

**AI ASSIGNMENTS**

**ASSIGNMENT – 5**

#include <bits/stdc++.h>

using namespace std;

struct Move {

int row, col;

};

// Evaluate the board for 'player'

int evaluate(vector<vector<char>> &board, char player) {

char opponent = (player == 'X') ? 'O' : 'X';

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

// Rows and Columns

if (board[i][0] == board[i][1] && board[i][1] == board[i][2]) {

if (board[i][0] == player) return 10;

if (board[i][0] == opponent) return -10;

}

if (board[0][i] == board[1][i] && board[1][i] == board[2][i]) {

if (board[0][i] == player) return 10;

if (board[0][i] == opponent) return -10;

}

}

// Diagonals

if (board[0][0] == board[1][1] && board[1][1] == board[2][2]) {

if (board[0][0] == player) return 10;

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;

if (board[0][2] == opponent) return -10;

}

return 0; // Draw or ongoing

}

// Get available moves

vector<Move> getAvailableMoves(vector<vector<char>> &board) {

vector<Move> moves;

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

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

if (board[i][j] == '.') moves.push\_back({i, j});

return moves;

}

// Move ordering: center > corners > edges

bool moveComparator(Move a, Move b) {

vector<pair<int,int>> priority = {

{1,1},{0,0},{0,2},{2,0},{2,2},{0,1},{1,0},{1,2},{2,1}

};

auto getPriority = [&](Move m){

for (int i = 0; i < priority.size(); i++)

if (priority[i].first == m.row && priority[i].second == m.col)

return i;

return 9;

};

return getPriority(a) < getPriority(b); // smaller index = higher priority

}

// Negamax with Alpha-Beta pruning and depth-limit

int negamax(vector<vector<char>> &board, char player, char opponent, int depth, int alpha, int beta) {

int score = evaluate(board, player);

if (score == 10 || score == -10 || depth == 0 || getAvailableMoves(board).empty())

return score;

int maxEval = INT\_MIN;

auto moves = getAvailableMoves(board);

sort(moves.begin(), moves.end(), moveComparator);

for (auto m : moves) {

board[m.row][m.col] = player;

int eval = -negamax(board, opponent, player, depth - 1, -beta, -alpha);

board[m.row][m.col] = '.';

maxEval = max(maxEval, eval);

alpha = max(alpha, eval);

if (alpha >= beta) break; // Prune

}

return maxEval;

}

// Find the best move with tie-breaking preference

Move findBestMove(vector<vector<char>> &board, char player, char opponent, int maxDepth) {

// Shortcut: if the center is empty at the start, choose it immediately

if (board[1][1] == '.') return {1, 1};

int bestVal = INT\_MIN;

Move bestMove = {-1, -1};

auto moves = getAvailableMoves(board);

sort(moves.begin(), moves.end(), moveComparator);

for (auto m : moves) {

board[m.row][m.col] = player;

int moveVal = -negamax(board, opponent, player, maxDepth - 1, INT\_MIN, INT\_MAX);

board[m.row][m.col] = '.';

// Prefer higher score; break ties by move priority

if (moveVal > bestVal || (moveVal == bestVal && moveComparator(m, bestMove))) {

bestVal = moveVal;

bestMove = m;

}

}

return bestMove;

}

int main() {

vector<vector<char>> board = {

{'.', '.', '.'},

{'.', '.', '.'},

{'.', '.', '.'}

};

char player = 'X', opponent = 'O';

Move best = findBestMove(board, player, opponent, 6); // depth-limited

cout << "Best Move: (" << best.row << ", " << best.col << ")\n";

return 0;

}

