Task 2.3: 7x7 Tic-tac-toe using alpha-beta pruning:

Heuristic:

No. of possible sequences to win in the next move.

Video Link:

https://youtu.be/JFG2lPm_Ecg

Code:

```
//Muhammad Somaan 2528404
//Muhammed Hashir Faraz 2528545

#include <stdio.h>
#include <time.h>
#include <ctype.h>
#include <malloc.h>

#define SIZE 7
#define DEPTH 7

struct TicTacToe
{
    char board[SIZE][SIZE];
    char playerChar;
    char AIChar;
};
typedef struct TicTacToe TicTacToe;

struct Move
{
    int x;
    int y;
};
typedef struct Move Move;


TicTacToe *createGame();
char getPlayerChar();

void playPlayerTurn(TicTacToe *game);

void playAITurn(TicTacToe* game);

Move* alphaBetaSearch(TicTacToe* game);
int max(TicTacToe* game, Move *move, int  alpha, int beta, int depth);
int min(TicTacToe* game, Move *move, int  alpha, int beta, int depth);

int checkWin(TicTacToe* game, char player);

int isBoardFull(TicTacToe* game);
int evaluateBoard(TicTacToe* game);
void printBoard(TicTacToe *game);

int checkBoardStatus(TicTacToe *game);



int main()
{
    TicTacToe *game = createGame();
```

```c
    int continueGame = 1;
    printBoard(game);

    while(continueGame)
    {
        playPlayerTurn(game);
        printBoard(game);
        continueGame = checkBoardStatus(game);

        if(!continueGame)
            break;

        clock_t start = clock();
        // Call the AlphaBet algorithm to calculate the next move
        playAITurn(game);
        clock_t end = clock();
        double elapsed_time = (double)(end - start) / CLOCKS_PER_SEC;
        printf("Elapsed time: %f\n", elapsed_time);

        printBoard(game);
        continueGame = checkBoardStatus(game);
    }

    printBoard(game);

    return 0;
}



TicTacToe *createGame()
{
    TicTacToe *game = (TicTacToe *) malloc(sizeof(TicTacToe));

    int i;
    int j;

    for(i=0; i<SIZE; i++)
        for(j=0; j<SIZE; j++)
            game->board[i][j] = '_';

    game->playerChar = getPlayerChar();
    game->AIChar = (game->playerChar == 'X') ? 'O' : 'X';

    return game;
}

char getPlayerChar()
{
    char playerChar = ' ';

    while (playerChar != 'X' && playerChar != 'O')
    {
        printf("Select your character X or O: ");
        scanf(" %c", &playerChar);
        playerChar = toupper(playerChar);
    }

    printf("You selected: %c\n", playerChar);
    return playerChar;
}
```

```c
void playPlayerTurn(TicTacToe *game)
{
    int row = -1;
    int col = -1;

    while (row < 0 || row >= SIZE || col < 0 || col >= SIZE || game-
>board[row][col] != '_')
    {
        printf("Enter the row and column for your move with no overlap eg:
(0 0): ");
        scanf(" %d %d", &row, &col);
    }

    game->board[row][col] = game->playerChar;
}


void playAITurn(TicTacToe* game)
{
    Move *move = alphaBetaSearch(game);
    game->board[move->x][move->y] = game->AIChar;
}


Move* alphaBetaSearch(TicTacToe* game)
{
    Move *move = (Move *) malloc(sizeof(Move));
    max(game, move, INT_MIN, INT_MAX, DEPTH);

    return move;
}

int max(TicTacToe* game, Move *move, int  alpha, int beta, int depth)
{
    int evaluation = evaluateBoard(game);
    if(evaluation != 0 || depth==0)
        return evaluation;
    else if(isBoardFull(game))
        return 0;

    int maxScore = INT_MIN;
    int bestMoveX = -1;
    int bestMoveY = -1;

    for (int i = 0; i < SIZE; i++)
    {
        for (int j = 0; j < SIZE; j++)
        {
            if (game->board[i][j] == '_')
            {
                game->board[i][j] = game->AIChar;
                int currentScore = min(game, move, alpha, beta, depth-1);

                //reverting the change back, so our original board is not
changed.
                game->board[i][j] = '_';

                if (currentScore > maxScore)
                {
                    bestMoveX = i;
```

```
                    bestMoveY = j;
                    maxScore = currentScore;

                    alpha = alpha>maxScore ? alpha : maxScore;
                }
                if(maxScore >= beta)
                {
                    move->x = bestMoveX;
                    move->y = bestMoveY;
                    return maxScore;
                }
            }
        }
    }

    move->x = bestMoveX;
    move->y = bestMoveY;

    return maxScore;
}

int min(TicTacToe* game, Move *move, int  alpha, int beta, int depth)
{
    int evaluation = evaluateBoard(game);
    if(evaluation != 0 || depth==0)
        return evaluation;
    else if(isBoardFull(game))
        return 0;

    int minScore = INT_MAX;

    for (int i = 0; i < SIZE; i++)
    {
        for (int j = 0; j < SIZE; j++)
        {
            if (game->board[i][j] == '_')
            {
                game->board[i][j] = game->playerChar;
                int currentScore = max(game, move, alpha, beta, depth-1);
                game->board[i][j] = '_';

                if (currentScore < minScore)
                {
                    minScore = currentScore;
                    beta = beta<minScore ? beta : minScore;
                }
                if(minScore <= alpha)
                    return minScore;
            }
        }
    }

    return minScore;
}

int checkWin(TicTacToe* game, char player)
{
    int wins = 0;
    //Checks board horizontally
    for (int x = 0; x < SIZE; x++)
    {
```

```
        for (int y = 0; y <= SIZE - 4; y++)
        {
            int horizontal = 1;

            for(int z = y; z < SIZE && z < y+4; z++)
                horizontal = horizontal && player == game->board[x][z];

            if (horizontal)
                wins++;
        }
    }

    //Checks board vertically
    for (int y = 0; y < SIZE; y++)
    {
        for (int x = 0; x <= SIZE - 4; x++)
        {
            int vertical = 1;

            for(int z = x; z < SIZE && z < x+4; z++)
                vertical = vertical && player == game->board[z][y];

            if (vertical)
                wins++;
        }
    }

    for (int x = 0; x < SIZE - 3; x++)
    {
        for (int y = 0; y < SIZE - 3; y++)
        {
            int diagonal = 1;
            for (int z = 0; z < 4; z++)
            {
                if (game->board[x + z][y + z] != player)
                {
                    diagonal = 0;
                    break;
                }
            }
            if (diagonal)
                wins++;
        }
    }

    // Check diagonally (top-right to bottom-left)
    for (int x = 0; x < SIZE - 3; x++)
    {
        for (int y = 3; y < SIZE; y++)
        {
            int diagonal1 = 1;
            for (int z = 0; z < 4; z++)
            {
                if (game->board[x + z][y - z] != player)
                {
                    diagonal1 = 0;
                    break;
                }
            }
            if (diagonal1)
                wins++;
```

```
        }
    }

    return wins;
}




int isBoardFull(TicTacToe* game)
{
    int i;
    int j;

    for (i = 0; i < SIZE; i++)
        for (j = 0; j < SIZE; j++)
            if (game->board[i][j] == '_')
                return 0;
    return 1;
}


int evaluateBoard(TicTacToe* game)
{
    int playerWins = checkWin(game, game->playerChar);
    int AIWins = checkWin(game, game->AIChar);
    if (AIWins)
        return AIWins; // Player wins
    else if (playerWins)
        return -1*playerWins; // AI wins
    else
        return 0; // Draw or in-progress
}

int checkBoardStatus(TicTacToe *game)
{
    if(isBoardFull(game))
    {
        printf("Its a draw!\n");
        return 0;
    }
    else
    {
        if(evaluateBoard(game) < 0)
        {
            printf("You won!\n");
            return 0;
        }
        else if(evaluateBoard(game) > 0)
        {
            printf("You lost!\n");
            return 0;
        }
    }

    return 1;
}


void printBoard(TicTacToe *game)
{
```

```
    printf("-------------\n");
    for (int i = 0; i < SIZE; i++)
    {
        for (int j = 0; j < SIZE; j++)
            printf("| %c ", game->board[i][j]);

        printf("|\n");
        printf("-------------\n\n");
    }
}
```

Outputs:

7x7 Board:

AI wins:

AI gets 4 consecutive “O” in diagonal and wins.