CS 5346- Project #2 Report

Adv Artificial Intelligence

Instructor:

Dr. Moonis Ali

Report By:

Vijay Samudrala

Group Members:

Hemanth Reddy Madduri

Sai Teja Mallidi

The Problem Description

Connect Four is a board game which contains six rows and seven columns. It is played between two players. Each player takes his turn in putting their symbol on the board (X or O).

The first player must place their symbol (X) in the 1st row. The second player must place their symbol (O) anywhere in the 1st row or exactly above the X in the 2nd row. This process repeats itself until the board is full I.e., the game is drawn or player wins.

A player wins the game when they can place their symbols in 4 continuous slots either horizontally, vertically, or diagonally.

We are using MINIMAXAB algorithm and AlphaBeta Search algorithms to find the best move for the player.

Each team member of the group is required to write an evaluation function that evaluates the next move.

Depending on the depth given by the user the next moves are predicted by these algorithms.

The game is played 9 times to determine the relation between different evaluation functions.

Usually the first turn MAX(algorithm, evalFunc) and the second turn is that of MIN(algorithm, evalFunc).

The Domain

The algorithms' main goal is to forecast the next optimal move depending on the depth provided by the users.

The depth represents the number of moves required for AI (Artificial Intelligence) to forecast and perceive all outcomes, trace the path to the best possible outcome, and decide based on that outcome.

To play this game, we will use a 6x7 board. A maximum of 21 times can be played by each player. If both players have completed their maximum number of turns and no one has connected four symbols, the game is considered a tie. A player wins if he can connect his symbol before the other player can.

Every time a move is made, the algorithm anticipates the next move by invoking the appropriate algorithm, which calculates future results based on the depth specified by the user, for example, if the depth is 2, the system checks every move for two levels of remaining play.

Methodologies

In this game we are using two different methods to compute the next move:

1. **MINIMAX-A-B:**
   MiniMax-A-B is an optimization technique for the MiniMax algorithm. This approach significantly decreases computation time. Because we do not need to search the full tree if we find a superior value, this technique allows us to search faster and move deeper into the game tree. This is known as Alpha Beta Pruning, and it adds two more parameters to the minimax algorithm: alpha and beta.

   At that level or higher, **alpha** is the best value that the maximizer can currently guarantee.

   At that level or higher, **beta** is the best value that the minimizer can currently guarantee.

   In the initial step –Infinity and +Infinity are passed to the algorithm and later as the algorithm proceeds the alpha, beta values are calculated and passed on recursively.

   Min and Max act as two different players at each level either Min is called, or Max is called depending on the turn of the player. Max tries to get the highest possible score while Min tries to get the lowest possible scores. Usually, Max takes the first turn.

   Utility functions are used to calculate the scores for each node with the help of leaves considering one layer at a time until the root of the tree. All of the backed-up values eventually reach the tree's root, or topmost point. MAX must choose the highest value at that stage.

2. **Alpha Beta Search**

   Alpha Beta search can be applied to any depth of the tree. This search algorithm helps us to terminate leaves or even sub trees based on the alpha and beta values.

   The main condition in alpha beta pruning is alpha>=beta. The max player will only update the value of alpha while min player can only update the value of beta. While backtracking to the root of the trees the calculated node valued is passed instead of the alpha and beta values. The alpha and beta values are passed to the child nodes only.

   In the first step max player will start the first move where alpha=-infinity and beta=+infinity. These values are passed all the way down to the root node and there depending on the turn the alpha or beta values are calculated and the node values are passed above to the parent node. So depending on the turn either max value or min value is selected.

   In some cases the alpha beta will not prune any of the subtrees and works exactly as minmax algorithm when this happens its takes more time than mini max because of the alpha and beta factors.

Program Implementation:

We are using a 2-D array for the board representation. The printBoard() function is used to print the status of the board after every move made. We keep track of the moves played by both the players using a vector pair of integers. The gameOver() functions takes the integer as an input which determines if the game is won by a player, or it is a draw.

The checkWin() functions is called to check if someone has won the game or not. It takes all the moves made by the player whose turn it is and the next move that the player wants to make, depending on the next move it decides if the player wins or not.
The deepEnough() function determines if we reached the depth that is given by the user.
The move() function returns the pair of integer of the next move.
The getValidMove() function returns the vector of pair of integers of all the possible moves a player can make.

The maxValue() and minValue() function determine the max and min value of the node.

The compute function is used to determine which evaluation function is to be used.

MinMaxAB and alphaBetaSearch are the algorithms used to determine the best move for the player.
The getAlphaBeta and getMiniMaxAB are the functions used to call the respective algorithms.
E1,e2,e3 are the three evaluation functions used to evaluate the status of the board.

Evaluation Functions:

In the first evaluation function depending on the possible moves list we assign scores to all the possible nodes and the node with the maximum or minimum value is selected as the best move.

For every value in the moves list, we check if there is a same symbol adjacent to it, we add 25 points to the score and if there is a different symbol, we subtract 25 points from it. So finally the node with highest score is the best move for the player.

We check for all the possible combinations for the symbols, vertically, horizontally and diagonally. If the slot is empty then we don't add or subtract any value from the current score.

We tend to check the symbols for only three consecutive slots because it takes only four continuous slots with same symbol to win the game. Checking all the slots to assign the scores is a lengthy process and is also time and memory consuming.

Source Code

ConnectFour.h

```
#include<iostream>
#include<vector>
using namespace std;
class Player {
public:
string b1[6][7];
bool turn;
int playerName;

int evalFunc1,evalFunc2;
int maxDepth;
int n1,n2,tn1,tn2;
int ne1,ne2,tne1,tne2;

vector<pair<int,int>> player1Moves;
vector<pair<int,int>> player2Moves;

Player();
void printBoard();
void resetBoard();
bool gameOver(int);
int checkWin(string);
int deepEnough(int);
pair<int,int> move();
```

```
vector<pair<int,int>> getValidMove();
int opposite(int);
pair<int, int> maxValue(int, int, int, int, int, pair<int,int>);
pair<int, int> minValue(int, int, int, int, int, pair<int,int>);

int compute(int, int, pair<int,int>);
pair<int,vector<pair<int,int>>> minMaxAB(pair<int,int>, int, int, int, int, int);
pair<int,int> getMinMaxAB(int, int, int, pair<int,int>);
pair<int, int> alphaBetaSearch(int, int, int, int, int,pair<int,int>);
pair<int,int> getAlphaBeta(int, int, int, pair<int,int>);

int e1(int, pair<int,int>);
int e2(int, pair<int,int>);
int e3(int, pair<int,int>);

};
```

ConnectFour.cpp

```
#include "connectFour.h"
#include<vector>
using namespace std;

Player::Player(){
turn = false;
playerName = 1;
```

```
n1 = n2 = 0;
evalFunc1 = evalFunc2 = 0;
tn1 = tn2 = 0;

for(int i=0; i<6; i++){
for(int j=0;j<7;j++){
b1[i][j] = " ";
}
}
}

void Player::printBoard(){
cout << " ----------------" << endl;
cout << "|" << b1[5][0] << "|" << b1[5][1] << "|" << b1[5][2] << "|" << b1[5][3]
<< "|" << b1[5][4] << "|" << b1[5][5] << "|" << b1[5][6] << "|" << endl;
cout << "|" << b1[4][0] << "|" << b1[4][1] << "|" << b1[4][2] << "|" << b1[4][3]
<< "|" << b1[4][4] << "|" << b1[4][5] << "|" << b1[4][6] << "|" << endl;
cout << "|" << b1[3][0] << "|" << b1[3][1] << "|" << b1[3][2] << "|" << b1[3][3]
<< "|" << b1[3][4] << "|" << b1[3][5] << "|" << b1[3][6] << "|" << endl;
cout << "|" << b1[2][0] << "|" << b1[2][1] << "|" << b1[2][2] << "|" << b1[2][3]
<< "|" << b1[2][4] << "|" << b1[2][5] << "|" << b1[2][6] << "|" << endl;
cout << "|" << b1[1][0] << "|" << b1[1][1] << "|" << b1[1][2] << "|" << b1[1][3]
<< "|" << b1[1][4] << "|" << b1[1][5] << "|" << b1[1][6] << "|" << endl;
cout << "|" << b1[0][0] << "|" << b1[0][1] << "|" << b1[0][2] << "|" << b1[0][3]
<< "|" << b1[0][4] << "|" << b1[0][5] << "|" << b1[0][6] << "|" << endl;
cout << " ----------------" << endl;
```

```
}

void Player::resetBoard(){
for(int i=0; i<6; i++){
for(int j=0;j<7;j++){
b1[i][j] = "";
}
}
}

bool Player::gameOver(int result){
if(result == 1)
return true;
else
return false;
}

int Player::checkWin(string currentPlayer){
//cout << currentPlayer << endl;
for(int i = 0; i < 6; i++){
for(int j = 0; j < 4; j++) {
if(b1[i][j] == currentPlayer){
//row
if(b1[i][j + 1] == currentPlayer && b1[i][j + 2] == currentPlayer && b1[i][j + 3]
== currentPlayer)
```

```
return 1;
}
}
}

for(int i = 0; i < 3; i++){
for(int j = 0; j < 7; j++) {
if(b1[i][j] == currentPlayer){
//column
if(b1[i + 1][j] == currentPlayer && b1[i + 2][j] == currentPlayer && b1[i + 3][j]
== currentPlayer)
return 1;
}
}
}

for(int i = 0; i < 3; i++){
for(int j = 0; j < 4; j++) {
if(b1[i][j] == currentPlayer){
//diagonal right
if(b1[i + 1][j + 1] == currentPlayer && b1[i + 2][j + 2] == currentPlayer &&
b1[i + 3][j + 3] == currentPlayer)
return 1;
}
}
```

```
}


for(int i = 0; i < 3; i++){
for(int j = 3; j < 7; j++) {
if(b1[i][j] == currentPlayer){
//diagonal left
if(b1[i + 1][j - 1] == currentPlayer && b1[i + 2][j - 2] == currentPlayer && b1[i
+ 3][j - 3] == currentPlayer)
return 1;
}
}
}


return 0;
}


vector<pair<int,int>> Player::getValidMove(){
vector<pair<int,int>> validMoves;
for(int i=0;i<5;i++){
for(int j=0;j<7;j++){
if(b1[i][j] == "X" || b1[i][j] == "O"){
if(b1[i+1][j] == " ")
validMoves.push_back({i+1,j});
}
if(b1[0][j] == " "){
```

```
validMoves.push_back({0,j});
}
}
}
return validMoves;
}

pair<int,int> Player::move(){
vector<pair<int,int>> validMoves = getValidMove();
return validMoves[0];
}

int Player::e1(int pname, pair<int,int> possibleMove){
int score = 0;
if(pname == 1){
//chechking row
for(int i=0; i<3; i++){
//checking forward
if(possibleMove.second < 4){
if(b1[possibleMove.first][possibleMove.second+i+1] == "X"){
score += 25;
}
}
else if(possibleMove.second == 4 && i!=2){
if(b1[possibleMove.first][possibleMove.second+i+1] == "X"){
```

```
score += 25;
}
}
else if(possibleMove.second == 5 && i<1){
if(b1[possibleMove.first][possibleMove.second+i+1] == "X"){
score += 25;
}
}
//checking backward
if(possibleMove.second > 2){
if(b1[possibleMove.first][possibleMove.second-i-1] == "X"){
score += 25;
}
}
else if(possibleMove.second == 2 && i!=2){
if(b1[possibleMove.first][possibleMove.second-i-1] == "X"){
score += 25;
}
}
else if(possibleMove.second == 1 && i<1){
if(b1[possibleMove.first][possibleMove.second-i-1] == "X"){
score += 25;
}
}
}
```

```
//checking column
for(int i=0; i<3; i++){
//checking forward
if(possibleMove.first < 3){
if(b1[possibleMove.first+i+1][possibleMove.second] == "X"){
score += 25;
}
}
else if(possibleMove.first == 3 && i!=2){
if(b1[possibleMove.first+i+1][possibleMove.second] == "X"){
score += 25;
}
}
else if(possibleMove.first == 4 && i<1){
if(b1[possibleMove.first+i+1][possibleMove.second] == "X"){
score += 25;
}
}
//checking backward
if(possibleMove.first > 2){
if(b1[possibleMove.first-i-1][possibleMove.second] == "X"){
score += 25;
}
}
```

```
else if(possibleMove.first == 2 && i!=2){
if(b1[possibleMove.first-i-1][possibleMove.second] == "X"){
score += 25;
}
}
else if(possibleMove.first == 1 && i<1){
if(b1[possibleMove.first-i-1][possibleMove.second] == "X"){
score += 25;
}
}
}

//checking diagnol
for(int i=0; i<3; i++){
//checking forward
if(possibleMove.first > 2 && possibleMove.second < 4){
if(b1[possibleMove.first-i-1][possibleMove.second+i+1] == "X"){
score += 25;
}
}
else if(possibleMove.first < 3  && possibleMove.second < 4){
if(b1[possibleMove.first+i+1][possibleMove.second+i+1] == "X"){
score += 25;
}
}
```

```
else if(possibleMove.first > 2 && possibleMove.second > 3){
if(b1[possibleMove.first-i-1][possibleMove.second-i-1] == "X"){
score += 25;
}
}
else if(possibleMove.first < 3 && possibleMove.second > 3){
if(b1[possibleMove.first+i+1][possibleMove.second-i-1] == "X"){
score += 25;
}
}
}
}
else if(pname == 2){
//chechking row
//cout << "Evaluation for Player 2" << endl;
for(int i=0; i<3; i++){
//checking forward
if(possibleMove.second < 4){
if(b1[possibleMove.first][possibleMove.second+i+1] == "O"){
score += 25;
}
}
else if(possibleMove.second == 4 && i!=2){
if(b1[possibleMove.first][possibleMove.second+i+1] == "O"){
score += 25;
```

```
}
}
else if(possibleMove.second == 5 && i<1){
if(b1[possibleMove.first][possibleMove.second+i+1] == "O"){
score += 25;
}
}
//checking backward
if(possibleMove.second > 2){
if(b1[possibleMove.first][possibleMove.second-i-1] == "O"){
score += 25;
}
}
else if(possibleMove.second == 2 && i!=2){
if(b1[possibleMove.first][possibleMove.second-i-1] == "O"){
score += 25;
}
}
else if(possibleMove.second == 1 && i<1){
if(b1[possibleMove.first][possibleMove.second-i-1] == "O"){
score += 25;
}
}
}
```

```
//checking column
for(int i=0; i<3; i++){
//checking forward
if(possibleMove.first < 3){
if(b1[possibleMove.first+i+1][possibleMove.second] == "O"){
score += 25;
}
}
else if(possibleMove.first == 3 && i!=2){
if(b1[possibleMove.first+i+1][possibleMove.second] == "O"){
score += 25;
}
}
else if(possibleMove.first == 4 && i<1){
if(b1[possibleMove.first+i+1][possibleMove.second] == "O"){
score += 25;
}
}
//checking backward
if(possibleMove.first > 2){
if(b1[possibleMove.first-i-1][possibleMove.second] == "O"){
score += 25;
}
}
else if(possibleMove.first == 2 && i!=2){
```

```
if(b1[possibleMove.first-i-1][possibleMove.second] == "O"){
score += 25;
}
}
else if(possibleMove.first == 1 && i<1){
if(b1[possibleMove.first-i-1][possibleMove.second] == "O"){
score += 25;
}
}
}

//checking diagnol
for(int i=0; i<3; i++){
//checking forward
if(possibleMove.first > 2 && possibleMove.second < 4){
if(b1[possibleMove.first-i-1][possibleMove.second+i+1] == "O"){
score += 25;
}
}
else if(possibleMove.first < 3  && possibleMove.second < 4){
if(b1[possibleMove.first+i+1][possibleMove.second+i+1] == "O"){
score += 25;
}
}
else if(possibleMove.first > 2 && possibleMove.second > 3){
```

```
if(b1[possibleMove.first-i-1][possibleMove.second-i-1] == "O"){
score += 25;
}
}
else if(possibleMove.first < 3 && possibleMove.second > 3){
if(b1[possibleMove.first+i+1][possibleMove.second-i-1] == "O"){
score += 25;
}
}
}
}
return score;
}

int Player::deepEnough(int d)
{
    if(d==maxDepth)
        return 0;
    else
return 1;
}

int Player::compute(int player, int eval, pair<int,int> pos){
if(eval==1)
    {
```

```
        return e1(player,pos);
    }
    else if(eval==2)
    {
        return e2(player, pos);
    }
    /*else
    {
        return e3(player);
    }*/
return e1(player,pos); //remove later
}

int Player::opposite(int player){
if(player == 0)
return 1;
else
return 2;
}
//Algorithms
pair<int,vector<pair<int,int>>> Player::minMaxAB(pair<int,int> position, int
depth, int player, int useThresh, int passThresh, int eFunc){
vector<pair<int,int>> successors;

if(player == 1){
```

```
n1++;
successors = getValidMove();
}
else{
n2++;
successors = getValidMove();
}


if(deepEnough(depth) || successors.size() == 0){
vector<pair<int,int>> path = successors;
if(player == 1)
return make_pair(-compute(player,eFunc,position),path);
else
return make_pair(compute(player,eFunc,position),path);
}

vector<pair<int,int>> bPath;
int newValue;
for (int i=0; i<successors.size(); i++){
pair<int,vector<pair<int,int>>> result =
minMaxAB(successors[i],depth+1,opposite(player), -useThresh, -
passThresh, eFunc);
newValue = -result.first;
if(newValue > passThresh){
```

```
passThresh = newValue;
bPath.push_back(successors[i]);
}
if(passThresh>=useThresh){
return(make_pair(passThresh,bPath));
}

}
return make_pair(passThresh,bPath);
}



pair<int,int> Player::getMinMaxAB(int depth, int evalFunc, int player,
pair<int,int> position){
maxDepth = depth;
if(player == 1){
n1=0;
ne1=0;
pair<int,vector<pair<int,int>>> mv =
minMaxAB(position,0,player,INT_MAX,INT_MIN,evalFunc);
tn1 += n1;
tne1 += ne1;
return mv.second.back();
}
else{
```

```
n2=0;
ne2=0;
pair<int,vector<pair<int,int>>> mv =
minMaxAB(position,0,player,INT_MAX,INT_MIN,evalFunc);
tn2 += n2;
tne2 += ne2;
return mv.second.back();
}
}

pair<int, int> Player::alphaBetaSearch(int depth, int playerName, int alpha,
int beta, int evalFunc, pair<int,int> pos){
return maxValue(depth, playerName, alpha, beta, evalFunc, pos);
}

pair<int, int> Player::maxValue(int depth, int playerName, int alpha, int beta,
int evalFunc, pair<int,int> pos){
vector<pair<int,int>> mv;
if(playerName == 1){
n1++;
mv = getValidMove();
}
else{
n2++;
mv = getValidMove();
```

```
}

if(depth == 0 || mv.size() == 0){
if(playerName == 1){
return make_pair(-compute(playerName, evalFunc,pos),-1);
}
else{
return make_pair(compute(playerName, evalFunc,pos),-1);
}
}

int v = INT_MIN;
pair<int,int> temp;
pair<int,int> val;
for(int i=0; i<mv.size(); i++){
val = minValue(depth-1, opposite(playerName), alpha, beta, evalFunc, pos);
v = max(v,val.first);
if(val.first == v){
temp = mv[i];
}
if(v >= beta){
return temp;
}
alpha = max(alpha,v);
if(alpha > v){
```

```
temp = mv[i];
}
}
return temp;
}

pair<int, int> Player::minValue(int depth, int playerName, int alpha, int beta,
int evalFunc, pair<int,int> pos){
vector<pair<int,int>> mv;
if(playerName == 1){
n1++;
mv = getValidMove();
}
else{
n2++;
mv = getValidMove();
}

if(depth == 0 || mv.size() == 0){
if(playerName == 1){
return make_pair(-compute(playerName, evalFunc, pos),-1);
}
else{
return make_pair(compute(playerName, evalFunc, pos),-1);
}
```

```
}

int v = INT_MAX;
pair<int,int> temp;
pair<int,int> val;
for(int i=0; i<mv.size(); i++){
val = maxValue(depth-1, opposite(playerName), alpha, beta, evalFunc, pos);
v = min(v,val.first);
if(val.first == v){
temp = mv[i];
}
if(v <= alpha){
return make_pair(v,temp.first);
}
beta = min(beta,v);
if(beta < v){
temp = mv[i];
}
}
return make_pair(v, temp.first);
}

pair<int,int> Player::getAlphaBeta(int depth, int playerName, int evalFunc,
pair<int,int> pos){
pair<int,int> mv;
```

```
mv = alphaBetaSearch(depth, playerName, INT_MIN, INT_MAX, evalFunc,
pos);
return mv;
}



/*int Player::e2(int pname, pair<int,int> possibleMove){
int utility = 138;
int score = 0;
string currentPlayer;
string nextPlayer;

if (pname == 0){
        currentPlayer = "O";
        nextPlayer = "X";
}
else {
        currentPlayer = "X";
        nextPlayer = "O";
}

int evaluationTable [6][7] = {{3, 4, 5, 7, 5, 4, 3},
                              {4, 6, 8, 10, 8, 6, 4},
                              {5, 8, 11, 13, 11, 8, 5},
                              {5, 8, 11, 13, 11, 8, 5},
```

```
                        {4, 6, 8, 10, 8, 6, 4},
                        {3, 4, 5, 7, 5, 4, 3}};

    for (int i = 0; i < 6; i++){
        for (int j = 0; j <7; j++){
            if (b1[i][j] == currentPlayer) score += evaluationTable[i][j];
            else if (b1[i][j] == nextPlayer) score -= evaluationTable[i][j];
        }
    }
score += evaluationTable[possibleMove.first][possibleMove.second];

    return utility + score;



}*/



//Evaluation Function 2
int Player::e2(int pname, pair<int,int> possibleMove){
int count = 0;
int score = 0;
string currentPlayer;
string nextPlayer;

if (pname == 0){
```

```
        currentPlayer = "O";
        nextPlayer = "X";
}
else {
        currentPlayer = "X";
        nextPlayer = "O";
}

int evalTable [6][7] = {{3, 4, 5, 7, 5, 4, 3},
                        {4, 6, 8, 10, 8, 6, 4},
                        {5, 8, 11, 13, 11, 8, 5},
                        {5, 8, 11, 13, 11, 8, 5},
                        {4, 6, 8, 10, 8, 6, 4},
                        {3, 4, 5, 7, 5, 4, 3}};

    for (int i = 0; i < 6; i++){
        for (int j = 0; j <7; j++){
            if (b1[i][j] == currentPlayer) score += evalTable[i][j];
            else if (b1[i][j] == nextPlayer) score -= evalTable[i][j];
        }
    }
int utility = 0;
for(int i = 1; i < 4; i++){
//column
int x = possibleMove.first + i;
```

```
int y = possibleMove.second;
if(x >= 0 && x < 6 && y >= 0 && y < 7 && b1[x][y] == currentPlayer) utility+=10;
if(x >= 0 && x < 6 && y >= 0 && y < 7 && b1[x][y] == nextPlayer) utility-=10;
x = possibleMove.first - i;
y = possibleMove.second;
if(x >= 0 && x < 6 && y >= 0 && y < 7 && b1[x][y] == currentPlayer) utility+=10;
if(x >= 0 && x < 6 && y >= 0 && y < 7 && b1[x][y] == nextPlayer) utility-=10;
//row
x = possibleMove.first;
y = possibleMove.second + i;
if(x >= 0 && x < 6 && y >= 0 && y < 7 && b1[x][y] == currentPlayer) utility+=10;
if(x >= 0 && x < 6 && y >= 0 && y < 7 && b1[x][y] == nextPlayer) utility-=10;
x = possibleMove.first;
y = possibleMove.second - i;
if(x >= 0 && x < 6 && y >= 0 && y < 7 && b1[x][y] == currentPlayer) utility+=10;
if(x >= 0 && x < 6 && y >= 0 && y < 7 && b1[x][y] == nextPlayer) utility-=10;
//diagonal right
x = possibleMove.first + i;
y = possibleMove.second + i;
if(x >= 0 && x < 6 && y >= 0 && y < 7 && b1[x][y] == currentPlayer) utility+=10;
if(x >= 0 && x < 6 && y >= 0 && y < 7 && b1[x][y] == nextPlayer) utility-=10;
```

```
x = possibleMove.first - i;
y = possibleMove.second - i;
if(x >= 0 && x < 6 && y >= 0 && y < 7 && b1[x][y] == currentPlayer)
utility+=10;
if(x >= 0 && x < 6 && y >= 0 && y < 7 && b1[x][y] == nextPlayer) utility-=10;
//diagonal left
x = possibleMove.first + i;
y = possibleMove.second - i;
if(x >= 0 && x < 6 && y >= 0 && y < 7 && b1[x][y] == currentPlayer)
utility+=10;
if(x >= 0 && x < 6 && y >= 0 && y < 7 && b1[x][y] == nextPlayer) utility-=10;
x = possibleMove.first - i;
y = possibleMove.second + i;
if(x >= 0 && x < 6 && y >= 0 && y < 7 && b1[x][y] == currentPlayer)
utility+=10;
if(x >= 0 && x < 6 && y >= 0 && y < 7 && b1[x][y] == nextPlayer) utility-=10;
}
score += evalTable[possibleMove.first][possibleMove.second];
//cout << utility + score << endl;
    return utility + score;
}
int Player::e3(int pname, pair<int,int> possibleMove){
int score = 0;
int i ,j;
int rows =6;
```

```
int cols = 7;
if(pname == 1){
for(i = 0;i<rows;i++){
for(j=0;j<cols-3;j++){
if(b1[possibleMove.first][possibleMove.second] == "X" &&
b1[possibleMove.first][possibleMove.second] ==
b1[possibleMove.first+i][possibleMove.second+1] &&
b1[possibleMove.first][possibleMove.second]==b1[possibleMove.first+i][pos
sibleMove.second+2]&&
b1[possibleMove.first][possibleMove.second]==b1[possibleMove.first+i][pos
sibleMove.second+3]){
score = 100;
}
}
}

for(i = 0;i< rows-3;i++){
for(j = 0;j<cols;j++){
if(b1[possibleMove.first][possibleMove.second] == "X" &&
b1[possibleMove.first][possibleMove.second] ==
b1[possibleMove.first+1][possibleMove.second+j] &&
b1[possibleMove.first][possibleMove.second]==b1[possibleMove.first+2][po
ssibleMove.second+j]&&
b1[possibleMove.first][possibleMove.second]==b1[possibleMove.first+3][po
ssibleMove.second+j]){
score = 100;
```

```
}
}
}
// diagonal positive win //
for (i = 0; i < rows - 3; i++)
        {
          for (j = 0; j< cols; j++)
            {
              if(b1[possibleMove.first][possibleMove.second+j] == "X" &&
b1[possibleMove.first][possibleMove.second+j] ==
b1[possibleMove.first+1][possibleMove.second+j+1] &&

b1[possibleMove.first][possibleMove.second]==b1[possibleMove.first+2][po
ssibleMove.second+j+2]&&
b1[possibleMove.first+i][possibleMove.second+j]==b1[possibleMove.first+3]
[possibleMove.second+j+3]){
score = 100;
}
          }
       }
  // diagonal negetative win //
  for (i = rows - 3; i <rows; i++){
          for (j = 0; j< cols - 3; j++){
              if(b1[possibleMove.first+i][possibleMove.second+j] == "X" &&
b1[possibleMove.first+i][possibleMove.second+j] == b1[possibleMove.first-
1][possibleMove.second+1] &&
```

```
b1[possibleMove.first+i][possibleMove.second+j]==b1[possibleMove.first-
2][possibleMove.second+2]&&
b1[possibleMove.first+i][possibleMove.second+j]==b1[possibleMove.first-
3][possibleMove.second+3]){
score = 100;
}
        }
}
}
if(pname == 2){
for(i = 0;i<rows;i++){
for(j=0;j<cols-3;j++){
if(b1[possibleMove.first+i][possibleMove.second+j] == "O" &&
b1[possibleMove.first+i][possibleMove.second+j] ==
b1[possibleMove.first+i][possibleMove.second+1] &&
b1[possibleMove.first+i][possibleMove.second+j]==b1[possibleMove.first+i][
possibleMove.second+2]&&
b1[possibleMove.first+i][possibleMove.second+j]==b1[possibleMove.first+i][
possibleMove.second+3]){
score = 100;
}
}
}


for(i = 0;i< rows-3;i++){
for(j = 0;j<cols;j++){
```

```
if(b1[possibleMove.first][possibleMove.second] == "O" &&
b1[possibleMove.first][possibleMove.second] ==
b1[possibleMove.first+1][possibleMove.second+j] &&
b1[possibleMove.first][possibleMove.second]==b1[possibleMove.first+2][po
ssibleMove.second+j]&&
b1[possibleMove.first][possibleMove.second]==b1[possibleMove.first+3][po
ssibleMove.second+j]){
score = 100;
}
}
}
// diagonal positive win //
for (i = 0; i < rows - 3; i++)
        {
          for (j = 0; j< cols; j++)
            {
              if(b1[possibleMove.first][possibleMove.second] == "O" &&
b1[possibleMove.first][possibleMove.second] ==
b1[possibleMove.first+1][possibleMove.second+1] &&
b1[possibleMove.first][possibleMove.second]==b1[possibleMove.first+2][po
ssibleMove.second+2]&&
b1[possibleMove.first][possibleMove.second]==b1[possibleMove.first+3][po
ssibleMove.second+3]){
score = 100;
}
            }
```

```
        }
  // diagonal negetative win //
  for (i = rows - 3; i <rows; i++)
        {
          for (j = 0; j< cols-3; j++)
            {
              if(b1[possibleMove.first][possibleMove.second] == "0" &&
b1[possibleMove.first][possibleMove.second] == b1[possibleMove.first-
1][possibleMove.second+1] &&
b1[possibleMove.first][possibleMove.second]==b1[possibleMove.first-
2][possibleMove.second+2]&&
b1[possibleMove.first][possibleMove.second]==b1[possibleMove.first-
3][possibleMove.second+3]){
score = 100;
}
            }
        }
}
return score;
}
```

Main.cpp

```
#include<iostream>
#include"connectFour.h"
#include<time.h>
```

```
using namespace std;



void procedure1(int ev1,int ev2,int d,Player p1){


int nm = 0;
vector<pair<int,int>> p1Moves = p1.player1Moves;
vector<pair<int,int>> p2Moves = p1.player2Moves;
if(nm == 0){
/*if(d>6){
int tp = d%7;
p1.b1[0][tp] = "X";
p1Moves.push_back({0,tp});
p1.printBoard();
p1.b1[0][tp+1] = "O";
p2Moves.push_back({0,tp+1});
p1.printBoard();
}
else{
p1.b1[0][d%7] = "X";
p1Moves.push_back({0,d});
p1.printBoard();
p1.b1[0][d+1] = "O";
p2Moves.push_back({0,d+1});
p1.printBoard();
```

```
}*/
int frstMv = rand()%7;
p1.b1[0][frstMv] = "X";
p1Moves.push_back({0,frstMv});
p1.printBoard();
p1.b1[1][frstMv] = "O";
p2Moves.push_back({1,frstMv});
p1.printBoard();
}
bool result = false;

while(!result){

if(nm == 42){
result == true;
cout << "It's a Draw!!" << endl;
}
else if(nm%2 == 0){
pair<int,int> temp = p1.getMinMaxAB(d,ev1,1,p1Moves.back());
p1.b1[temp.first][temp.second] = "X";
p1.printBoard();
//int won = p1.checkWin(p1Moves,temp);
int won = p1.checkWin("X");
p1Moves.push_back(temp);
result = p1.gameOver(won);
```

```
if(result == true)
cout << "Player 1 won the game." << endl;
}
else{
pair<int,int> temp = p1.getAlphaBeta(d,2,ev2, p2Moves.back());
p1.b1[temp.first][temp.second] = "O";
p1.printBoard();
//int won = p1.checkWin(p2Moves,temp);
int won = p1.checkWin("O");
p2Moves.push_back(temp);
result = p1.gameOver(won);
if(result == true)
cout << "Player 2 won the game." << endl;
}
nm++;
//cout << "Result: "<< result << endl;
}
}

void procedure2(int ev1,int ev2,int d,Player p1){

int nm = 0;
vector<pair<int,int>> p1Moves = p1.player1Moves;
vector<pair<int,int>> p2Moves = p1.player2Moves;
if(nm == 0){
```

```
int frstMv = rand() % 7;
p1.b1[0][frstMv] = "X";
p1Moves.push_back({0,frstMv});
p1.printBoard();
p1.b1[1][frstMv] = "O";
p2Moves.push_back({1,frstMv});
p1.printBoard();
}
bool result = false;

while(!result){
if(nm == 42){
result == true;
cout << "It's a Draw!!" << endl;
}
if(nm%2 == 0){
pair<int,int> temp = p1.getMinMaxAB(d,ev1,1,p2Moves.back());
p1.b1[temp.first][temp.second] = "X";
p1.printBoard();
//int won = p1.checkWin(p1Moves,temp);
int won = p1.checkWin("X");
p1Moves.push_back(temp);
result = p1.gameOver(won);
if(result == true)
cout << "Player 1 won the game." << endl;
```

```
}
else{
pair<int,int> temp = p1.getMinMaxAB(d,ev2,1, p1Moves.back());
p1.b1[temp.first][temp.second] = "O";
p1.printBoard();
//int won = p1.checkWin(p2Moves,temp);
int won = p1.checkWin("O");
p2Moves.push_back(temp);
result = p1.gameOver(won);
if(result == true)
cout << "Player 2 won the game." << endl;
}
nm++;
}
}

void procedure3(int ev1,int ev2,int d,Player p1){

int nm = 0;
vector<pair<int,int>> p1Moves = p1.player1Moves;
vector<pair<int,int>> p2Moves = p1.player2Moves;
if(nm == 0){
int frstMv = rand() % 7;
p1.b1[0][frstMv] = "X";
p1Moves.push_back({0,frstMv});
```

```
p1.printBoard();
p1.b1[1][frstMv] = "O";
p2Moves.push_back({1,frstMv});
p1.printBoard();
}
bool result = false;

while(!result){
if(nm == 42){
result == true;
cout << "It's a Draw!!" << endl;
}
if(nm%2 == 0){
pair<int,int> temp = p1.getAlphaBeta(d,1,ev1,p1Moves.back());
p1.b1[temp.first][temp.second] = "X";
p1.printBoard();
//int won = p1.checkWin(p1Moves,temp);
int won = p1.checkWin("X");
p1Moves.push_back(temp);
result = p1.gameOver(won);
if(result == true)
cout << "Player 1 won the game." << endl;
}
else{
pair<int,int> temp = p1.getAlphaBeta(d,2,ev2, p2Moves.back());
```

```
p1.b1[temp.first][temp.second] = "O";
p1.printBoard();
//int won = p1.checkWin(p2Moves,temp);
int won = p1.checkWin("O");
p2Moves.push_back(temp);
result = p1.gameOver(won);
if(result == true)
cout << "Player 2 won the game." << endl;
}
nm++;
}
}


int main(){
srand(time(0));
Player p1;
int coordinates = 0;
cout << "Let's Play the Game of Connect 4!" << endl;
cout<<" ------------------------------------------------------------\n";
    cout<<"| 1.Max(MinmaxAB) vs Min(AlphaBetaSearch) eval->1,1        |\n";
    cout<<"| 2.Max(MinmaxAB) vs Min(AlphaBetaSearch) eval->2,2        |\n";
    cout<<"| 3.Max(MinmaxAB) vs Min(AlphaBetaSearch) eval->3,3        |\n";
    cout<<"| 4.Max(MinmaxAB) vs Min(MinmaxAB) eval->1,2               |\n";
    cout<<"| 5.Max(MinmaxAB) vs Min(MinmaxAB) eval->1,3               |\n";
```

```
   cout<<"| 6.Max(MinmaxAB) vs Min(MinmaxAB) eval->2,3              |\n";
   cout<<"| 7.Max(AlphabetaSearch) vs Min(AlphaBetaSearch) eval->1,2
|\n";
   cout<<"| 8.Max(AlphabetaSearch) vs Min(AlphaBetaSearch) eval->1,3
|\n";
   cout<<"| 9.Max(AlphaBetaSearch) vs Min(AlphaBetaSearch) eval->2,3
|\n";
   cout<<" ------------------------------------------------------------\n";
int d,c;
cout << "Enter Choice: ";
cin >> c;
cout << endl;
cout << "Enter depth: ";
cin >> d;
cout << endl;


if(c==1)
   {
      cout<<"******* Starting the Game: ******* \n"<<"PLAYER 1:
Max(MinmaxAB) vs PLAYER 2: Min(AlphaBetaSearch)"<<endl;
      cout<<"Player1 evaluation function: "<< 1 <<" and Player2 evaluation
function: "<< 1 <<" and Depth: "<<d <<endl;
      cout<<"-------------------------------------------------------------------------
----------\n";
      //intial state of the board..
      cout<<"Initial state of the board"<<endl;
```

```
        p1.printBoard();
        p1.maxDepth=d;
        p1.playerName=1;
        p1.evalFunc1=1;
        p1.evalFunc2=1;
        procedure1(p1.evalFunc1, p1.evalFunc2, d, p1);



        cout<<"-------------------------------------------------------------------------------
-----------\n";
    }
else if (c==2){
cout<<"******* Starting the Game: ******* \n"<<"PLAYER 1:
Max(MinmaxAB) vs PLAYER 2: Min(AlphaBetaSearch)"<<endl;
        cout<<"Player1 evaluation function: "<< 2 <<" and Player2 evaluation
function: "<< 2 <<" and Depth: "<<d <<endl;
        cout<<"-------------------------------------------------------------------------------
-----------\n";
        //intial state of the board..
        cout<<"Initial state of the board"<<endl;
        p1.printBoard();
        p1.maxDepth=d;
        p1.playerName=1;
        p1.evalFunc1=2;
        p1.evalFunc2=2;
        procedure1(p1.evalFunc1, p1.evalFunc2, d, p1);
```

```
	cout<<"--------------------------------------------------------------------------------
----------\n";
}
else if (c==3){
cout<<"******* Starting the Game: ******* \n"<<"PLAYER 1:
Max(MinmaxAB) vs PLAYER 2: Min(AlphaBetaSearch)"<<endl;
	cout<<"Player1 evaluation function: "<< 3 <<" and Player2 evaluation
function: "<< 3 <<" and Depth: "<<d <<endl;
	cout<<"--------------------------------------------------------------------------------
----------\n";
	//intial state of the board..
	cout<<"Initial state of the board"<<endl;
	p1.printBoard();
	p1.maxDepth=d;
	p1.playerName=1;
	p1.evalFunc1=3;
	p1.evalFunc2=3;
	procedure1(p1.evalFunc1, p1.evalFunc2, d, p1);



	cout<<"--------------------------------------------------------------------------------
----------\n";
}
else if (c==4){
```

```
cout<<"******* Starting the Game: ******* \n"<<"PLAYER 1:
Max(MinmaxAB) vs PLAYER 2: Min(MinmaxAB)"<<endl;

    cout<<"Player1 evaluation function: "<< 1 <<" and Player2 evaluation
function: "<< 2 <<" and Depth: "<<d <<endl;

    cout<<"-------------------------------------------------------------------------
----------\n";

    //intial state of the board..

    cout<<"Initial state of the board"<<endl;

    p1.printBoard();

    p1.maxDepth=d;

    p1.playerName=1;

    p1.evalFunc1=1;

    p1.evalFunc2=2;

    procedure2(p1.evalFunc1, p1.evalFunc2, d, p1);



    cout<<"-------------------------------------------------------------------------
----------\n";

}

else if (c==5){

cout<<"******* Starting the Game: ******* \n"<<"PLAYER 1:
Max(MinmaxAB) vs PLAYER 2: Min(MinmaxAB)"<<endl;

    cout<<"Player1 evaluation function: "<< 1 <<" and Player2 evaluation
function: "<< 3 <<" and Depth: "<<d <<endl;

    cout<<"-------------------------------------------------------------------------
----------\n";

    //intial state of the board..
```

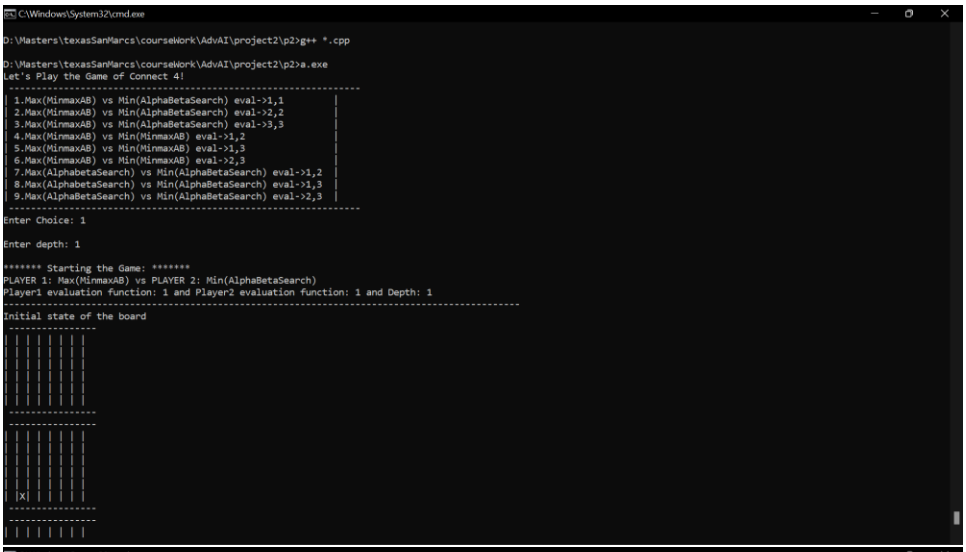
```
        cout<<"Initial state of the board"<<endl;
        p1.printBoard();
        p1.maxDepth=d;
        p1.playerName=1;
        p1.evalFunc1=1;
        p1.evalFunc2=3;
        procedure2(p1.evalFunc1, p1.evalFunc2, d, p1);



        cout<<"-------------------------------------------------------------------------------
-----------\n";
}
else if (c==6){
cout<<"******* Starting the Game: ******* \n"<<"PLAYER 1:
Max(MinmaxAB) vs PLAYER 2: Min(MinmaxAB)"<<endl;
        cout<<"Player1 evaluation function: "<< 2 <<" and Player2 evaluation
function: "<< 3 <<" and Depth: "<<d <<endl;
        cout<<"-------------------------------------------------------------------------------
-----------\n";
        //intial state of the board..
        cout<<"Initial state of the board"<<endl;
        p1.printBoard();
        p1.maxDepth=d;
        p1.playerName=1;
        p1.evalFunc1=2;
        p1.evalFunc2=3;
```

```
        procedure2(p1.evalFunc1, p1.evalFunc2, d, p1);



        cout<<"-------------------------------------------------------------------------------
-----------\n";
}
else if (c==7){
cout<<"******* Starting the Game: ******* \n"<<"PLAYER 1:
Max(MinmaxAB) vs PLAYER 2: Min(AlphaBetaSearch)"<<endl;
        cout<<"Player1 evaluation function: "<< 1 <<" and Player2 evaluation
function: "<< 2 <<" and Depth: "<<d <<endl;
        cout<<"-------------------------------------------------------------------------------
-----------\n";
        //intial state of the board..
        cout<<"Initial state of the board"<<endl;
        p1.printBoard();
        p1.maxDepth=d;
        p1.playerName=1;
        p1.evalFunc1=1;
        p1.evalFunc2=2;
        procedure3(p1.evalFunc1, p1.evalFunc2, d, p1);



        cout<<"-------------------------------------------------------------------------------
-----------\n";
}
```

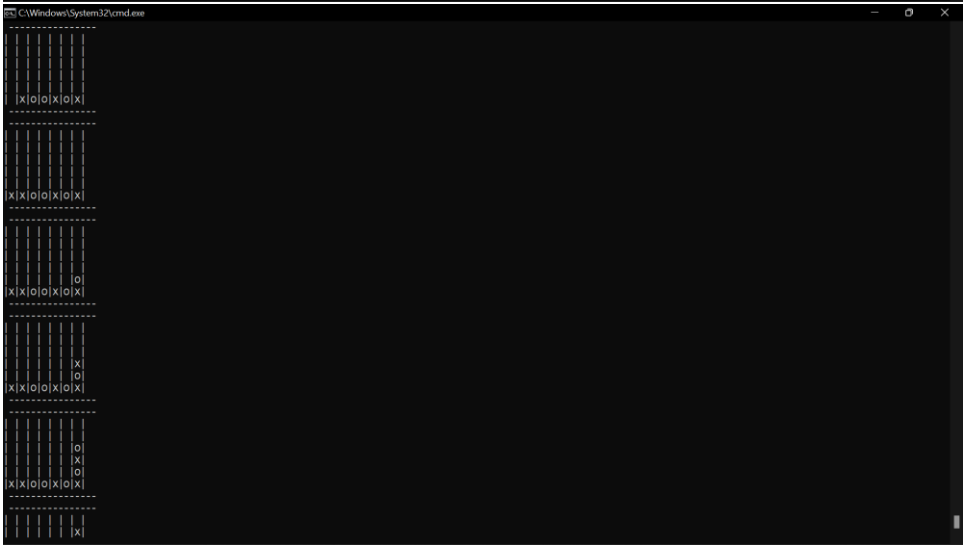
```
else if (c==8){
cout<<"******* Starting the Game: ******* \n"<<"PLAYER 1: Max(MinmaxAB) vs PLAYER 2: Min(AlphaBetaSearch)"<<endl;
    cout<<"Player1 evaluation function: "<< 1 <<" and Player2 evaluation function: "<< 3 <<" and Depth: "<<d <<endl;
    cout<<"--------------------------------------------------------------------------------\n";
    //intial state of the board..
    cout<<"Initial state of the board"<<endl;
    p1.printBoard();
    p1.maxDepth=d;
    p1.playerName=1;
    p1.evalFunc1=1;
    p1.evalFunc2=3;
    procedure3(p1.evalFunc1, p1.evalFunc2, d, p1);
}
else if (c==9){
cout<<"******* Starting the Game: ******* \n"<<"PLAYER 1: Max(MinmaxAB) vs PLAYER 2: Min(AlphaBetaSearch)"<<endl;
    cout<<"Player1 evaluation function: "<< 2 <<" and Player2 evaluation function: "<< 3 <<" and Depth: "<<d <<endl;
    cout<<"--------------------------------------------------------------------------------\n";
    //intial state of the board..
    cout<<"Initial state of the board"<<endl;
    p1.printBoard();
```

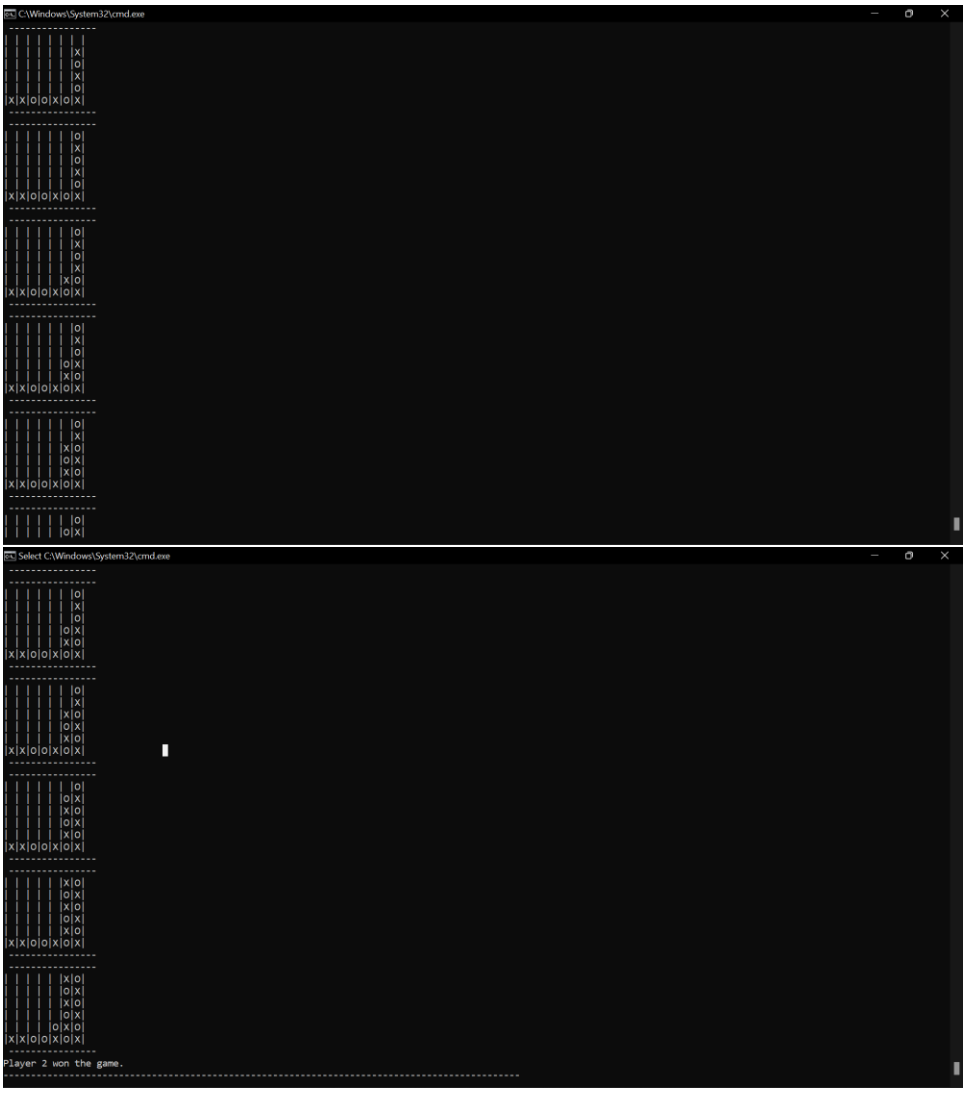
```
        p1.maxDepth=d;
        p1.playerName=1;
        p1.evalFunc1=2;
        p1.evalFunc2=3;
        procedure3(p1.evalFunc1, p1.evalFunc2, d, p1);
}



return 0;
}
```

A Copy of the program Run


C:\Windows\System32\cmd.exe
D:\Masters\texasSanMarcs\courseWork\AdvAI\project2\p2>g++ *.cpp
D:\Masters\texasSanMarcs\courseWork\AdvAI\project2\p2>a.exe
Let's Play the Game of Connect 4!
1.Max(MinmaxAB) vs Min(AlphaBetaSearch) eval->1,1
2.Max(MinmaxAB) vs Min(AlphaBetaSearch) eval->2,2
3.Max(MinmaxAB) vs Min(AlphaBetaSearch) eval->3,3
4.Max(MinmaxAB) vs Min(MinmaxAB) eval->1,2
5.Max(MinmaxAB) vs Min(MinmaxAB) eval->1,3
6.Max(MinmaxAB) vs Min(MinmaxAB) eval->2,3
7.Max(AlphabetaSearch) vs Min(AlphaBetaSearch) eval->1,2
8.Max(AlphabetaSearch) vs Min(AlphaBetaSearch) eval->1,3
9.Max(AlphaBetaSearch) vs Min(AlphaBetaSearch) eval->2,3
Enter Choice: 1
Enter depth: 1
****** Starting the Game: *******
PLAYER 1: Max(MinmaxAB) vs PLAYER 2: Min(AlphaBetaSearch)
Player1 evaluation function: 1 and Player2 evaluation function: 1 and Depth: 1
Initial state of the board
|X|


C:\Windows\System32\cmd.exe
X	O
X	O
X	O
X	O	X	O	X
X	O	O	X	O	X


C:\Windows\System32\cmd.exe
X	O	O	X	O	X
X	X	O	O	X	O	X
X	X	O	O	X	O	X
X	X	O	O	X	O	X
X	X	O	O	X	O	X



Analysis of the program

Using a 2-D array as the board helps us to navigate through the board easily. The use of pairs that determine the coordinates on the board helps us efficiently know the exact position where the next player is going to place their symbol.

The minimaxAB and alphaBetaSearch algorithms are used to find the best path that is to be taken by the player to win the game. Essentially both the algorithms do the same work but in minimaxAB we are using the threshold values to determine the next move whereas in alpha beta search technique we use alpha and beta values that are modified continuously at each level of the tree.

The evaluation function returns an integer value which determines the score that is given to each node. Depending on the scores assigned by the evaluation function we see which node is the best possible move that the player can make to win the game or to block the opponent from winning the game.

The moves played by each player are stored in a vector pair of integers, this gives us the flexibility to keep track of moves in the order the player has played the move. We can assess them in a similar fashion as we access the elements in arrays.

The user enters the depth and the choice essentially the type of evaluation function and algorithm that they want to execute. Depending on the choice entered by the user the compute function call the respected evaluation function that needs to be executed for that player. The first move for both the players is generated randomly.

Analysis of the results

Choice	Length of Game Path	Nodes Generated	Nodes Expanded	Memory Used	Winning Percentage
1	~6	30	8	1mb	40% player 1
2	~5	27	9	1.2mb	42% player 1
3	~4	35	10	1 mb	50% player 2
4	~6	20	12	1.5mb	30% player 1
5	~4	29	10	1.2 mb	70% player 1
6	~7	22	6	1 mb	50 % player 2
7	~4	30	10	1.2 mb	60% player 2
8	~4	25	7	1 mb	80 % player 1
9	~10	48	15	2 mb	75% player 2

Conclusion

From the results we can conclude that as the depth increases the memory used by the program, time taken to execute also increase.

Alpha beta pruning paves the way to develop more advanced approaches for solving problems. The alpha-beta pruning algorithm is a quality

optimization over the min-max algorithm, and together with the minimax algorithm is becoming the foundation of the searching techniques.

The benefit of alpha–beta pruning is that it allows you to remove branches from the search tree. This way, the search time can be reduced to the more promising subtree while also performing a deeper search. It belongs to the branch and bound class of algorithms, just like its predecessor.

Using ordering heuristics to search earlier areas of the tree that are likely to trigger alpha–beta cutoffs can enhance performance without losing accuracy.

References

- https://stackoverflow.com/questions/36892813/minimax-algorithm-advantages-disadvantages

- https://www.educative.io/edpresso/what-is-alpha-beta-pruning
- https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning
- https://en.wikipedia.org/wiki/Connect_Four
- https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/

Contributions

Hemanth Reddy:

-Evaluation Function 2

-Developing checkWin, AlphaBetaSearch functions

-Testing CheckWin, minimaxAB

SaiTeja:

-Evaluation function 3

-Developing possibleMoves, MinimaxAB

-Testing AlphaBetaSearch

Vijay

-Evaluation function1

-Developing alphaBetaSerch, MinimaxAB, possibleMoves

-Unit testing and integrating the code