CS4346 AI Project 2:

TICTACTOE with MinMax Algorithm with Alpha Beta Pruning

By Yuvanesh Rajamani

**Problem description:**

We are using MinMax-A-B in our TicTacToe game. We will be using four different evaluation function to see how they affect the results of the game.

**Domain:**

Our goal is to create a TicTacToe game in which both players will be using the MinMax-A-B algorithm to determine their best move and play it. The moves will be calculated using our evaluation function that we developed for this problem. We will run the program several times with the players using different evaluation functions to see how the results varies/

**Methodologies:**

MinMax:

MinMax is a backwards tracking algorithm that works by assuming that the player and opponent will both make the move that is best for them. The maximizer will try to get the highest score while the minimizer will aim for the lowest score. In our TicTacToe game the algorithm will start at the initial state. Then it will determine whose move it is. The algorithm will recursively call itself, alternating between players. The algorithm will work itself down the game tree, generating all possible moves for each player(leafs). Once the algorithm works its way down to the last ply, the algorithm will get the best move from the generated nodes for each player and back it up the tree. This will occur at each ply until your back at the initial state then you will have the next best move for the player. This will occur for each turn.

MinMax-A-B:

Alpha-Beta search build upon the minmax algorithm described above. A drawback of the minmax algorithm is that it is time and space consuming since it will generate and expand every node possible in a game. Alpha-Beta simplifies this by creating an alpha value that is associated with the max node and a beta value that is associated with the min node. These are values that the algorithm will compare against to determine if a node path is even worth expanding. If the node fails the corresponding comparison then the node won’t be expanded and the entire path will be pruned which will result in a faster algorithm and less memory since there’s less nodes to search.

**Source Code Implementation:**

We built our program by using the MinMax-A-b pseudocode provided by Dr.Moonis Ali. Inorder to make the algorithm work as a game played between two AIs we had the make some alterations with the pseudocode. The first one being that we are not returning a path instead we are just returning a number that represents the next best move. This is so the game can be played continuously between the two Ai players. The next change is when we are calling the evaluation function. The function is called at the beginning of the minMax algorithm in our gameover function this is so we can get the board value at the start of the function call.

**Source Code:**

Main.cpp:

*// Artificial Intelligence CS 4346.001  
// Project 2 MinMax algorithm with alpha beta pruning  
// Team Members: Yuvanesh Rajamani, Nico Maldonado, John Edwards.*#include <iostream>  
#include <cstdlib>  
#include <limits.h>  
#include <unistd.h>  
#include <sys/types.h>  
#include <fstream>  
#include <vector>  
  
  
using namespace std;  
*//Array for the board*char board[3][3] = {{'1','2','3'},{'4','5','6'},{'7','8','9'}};  
*//Variable Declaration*int choice;  
int row,column;  
char turn = 'X';  
bool draw = false;  
  
int miniMax(char board[3][3], int depth, char player, int alpha, int beta);  
int bestRow= 0;  
int bestCol = 0;  
int allTries = 0;  
int allTriesP = 0;  
int totalTries = 0;  
int totalTriesP = 0;  
int totalTriesX = 0;*// nodes generated after pruning for x*int totalTriesO = 0;*// nodes generated after pruning for x*int totalTriesXbP = 0; *// nodes generated before pruning for x*int totalTriesObP = 0;*// nodes generated before pruning for o*bool xWin = false;  
bool oWin = false;  
bool gameover();  
int totalTime1 = 0;  
int totalTime2 = 0;  
*//Function to show the current status of the gaming board*void display\_board(){  
  
 *//Rander Game Board LAYOUT* cout<<"PLAYER - 1 [X] PLAYER - 2 [O]"<< endl;  
 cout<<" | | " << endl;  
 cout<<" "<<board[0][0]<<" | "<<board[0][1]<<" | "<<board[0][2]<<" " << endl;  
 cout<<"\_\_\_\_\_|\_\_\_\_\_|\_\_\_\_\_" << endl;  
 cout<<" | | " << endl;  
 cout<<" "<<board[1][0]<<" | "<<board[1][1]<<" | "<<board[1][2]<<" " << endl;  
 cout<<"\_\_\_\_\_|\_\_\_\_\_|\_\_\_\_\_" << endl;  
 cout<<" | | " << endl;  
 cout<<" "<<board[2][0]<<" | "<<board[2][1]<<" | "<<board[2][2]<<" " << endl;  
 cout<<" | | " << endl;  
}  
  
*//Function to get the player input and update the board*void bestMove();  
void player\_turn(){  
 if(turn == 'X'){  
 cout<<"Player - 1 [X] turn : ";  
 int start\_s = clock();  
 bestMove();  
 int stop\_s = clock();  
 totalTime1 += (stop\_s-start\_s)/double(CLOCKS\_PER\_SEC)\*1000;  
 cout << allTries << " Nodes Generated" << endl;  
 totalTries += allTries;  
 totalTriesXbP += allTries;  
 allTries = 0;  
 cout << allTriesP << " Nodes Generated after Prune" << endl;  
 totalTriesP += allTriesP;  
 totalTriesX +=allTriesP;  
 allTriesP = 0;  
  
  
 }  
 else if(turn == 'O'){  
 cout<<"Player - 2 [O] turn : ";  
 int start\_s = clock();  
 bestMove();  
 int stop\_s = clock();  
 totalTime2 += (stop\_s-start\_s)/double(CLOCKS\_PER\_SEC)\*1000;  
 cout << allTries << " Nodes Generated" << endl;  
 totalTries += allTries;  
 totalTriesObP += allTries;  
 allTries = 0;  
 cout << allTriesP << " Nodes Generated after Prune" << endl;  
 totalTriesP += allTriesP;  
 totalTriesO +=allTriesP;  
 allTriesP = 0;  
 }  
}  
  
  
  
void bestMove(){  
 int bestScore = -INT\_MAX;  
 int score = 0;  
 if(turn == 'X') {  
 for (int i = 0; i < 3; i++) {  
 for (int j = 0; j < 3; j++) {  
 if (board[i][j] != 'X' && board[i][j] != 'O') {  
 char temp = board[i][j];  
 board[i][j] = 'X';  
 score = miniMax(board, 0, 'O', INT\_MIN, INT\_MAX);  
 board[i][j] = temp;  
 if (score > bestScore) {  
 bestScore = score;  
 bestRow = i;  
 bestCol = j;  
 }  
 }  
 }  
 }  
 cout << bestScore << endl;  
 board[bestRow][bestCol] = 'X';  
 turn = 'O';  
 }else if(turn == 'O'){  
 for (int i = 0; i < 3; i++) {  
 for (int j = 0; j < 3; j++) {  
 if (board[i][j] != 'X' && board[i][j] != 'O') {  
 char temp = board[i][j];  
 board[i][j] = 'O';  
 score = miniMax(board, 0, 'X', INT\_MIN, INT\_MAX);  
 board[i][j] = temp;  
 if (score > bestScore) {  
 bestScore = score;  
 bestRow = i;  
 bestCol = j;  
 }  
 }  
 }  
 }  
 cout << bestScore << endl;  
 board[bestRow][bestCol] = 'O';  
 turn = 'X';  
 }  
 xWin =false;  
 oWin = false;  
 draw = false;  
}  
  
*// base evaluation shown in class*int evaluate1(char board[3][3], char player, char opp){  
 int maxi =0;  
 int mini =0;  
  
 char opponent = opp;  
 for(int row =0; row < 3; row++){  
 if(board[row][0]!= opponent && board[row][1]!= opponent && board[row][2]!= opponent ){  
 maxi++;  
 }  
 if(board[row][0]!= player && board[row][1]!= player && board[row][2]!= player){  
 mini++;  
 }  
 }  
 for(int col =0; col<3; col++){  
 if(board[0][col]!= opponent && board[1][col]!= opponent && board[2][col]!= opponent ){  
 maxi++;  
 }  
 if(board[0][col]!= player && board[1][col]!= player && board[2][col]!= player){  
 mini++;  
 }  
 }  
 if(board[0][0]!= opponent && board[1][1]!= opponent && board[2][2]!= opponent ){  
 maxi++;  
 }  
 if(board[0][0]!= player && board[1][1]!= player && board[2][2]!= player){  
 mini++;  
 }  
 if(board[0][2]!= opponent && board[1][1]!= opponent && board[2][0]!= opponent ){  
 maxi++;  
 }  
 if(board[0][2]!= player && board[1][1]!= player && board[2][0]!= player){  
 mini++;  
 }  
  
 return maxi-mini;  
}  
  
*// Nicos practice evaluation not used in program runs*int evaluate(char player){  
 if(player == 'X') {  
 if (xWin) {  
 xWin = false;  
 return 1;  
 } else if (oWin) {  
 oWin = false;  
 return -1;  
 } else {  
 return 0;  
 }  
 }else if(player == 'O'){  
 if (oWin) {  
 oWin = false;  
 return 1;  
 } else if (xWin) {  
 xWin = false;  
 return -1;  
 } else {  
 return 0;  
 }  
 }  
}  
  
*// Nicos evaluation*int evaluate2(char board[3][3], char player, int depth){  
 if(player == 'X') {  
 if (xWin) {  
 xWin = false;  
 for (int i = 0; i < 9; i++) {  
 if (depth == i) {  
 return 11 + i;  
 }  
 }  
 } else if (oWin) {  
 oWin = false;  
 for (int i = 0; i < 9; i++) {  
 if (depth == i) {  
 return -11 - i;  
 }  
 }  
 } else {  
 return 0;  
 }  
 }else if(player == 'O'){  
 if (oWin) {  
 oWin = false;  
 for (int i = 0; i < 9; i++) {  
 if (depth == i) {  
 return 11 + i;  
 }  
 }  
 } else if (xWin) {  
 xWin = false;  
 for (int i = 0; i < 9; i++) {  
 if (depth == i) {  
 return -11 - i;  
 }  
 }  
 } else {  
 return 0;  
 }  
 }  
}  
 *// Johns evaluation*int evaluate3(char board[3][3], char player, char opp) {  
 int MS\_board[3][3] = {{8, 3, 4}, {1, 5, 9}, {6, 7, 2}};  
 vector<int> moves\_p, moves\_o, winMoves, lossMoves;  
  
 *//accumulating the values for each player's moves* for (int i = 0; i < 3; i++) {  
 for (int j = 0; j < 3; j++) {  
 if (board[i][j] == player) {  
 moves\_p.push\_back(MS\_board[i][j]);  
 }  
 if (board[i][j] == opp) {  
 moves\_o.push\_back(MS\_board[i][j]);  
 }  
 }  
 }  
  
 *//accumlating winning moves* if (moves\_p.size() > 2) {  
 for (int i = 0; i < moves\_p.size()-1; i++) {  
 for (int j = i+1; j < moves\_p.size(); j++) {  
 int m = 15 - (moves\_p[i] + moves\_p[j]);  
 if (m > 0 && m < 10) {  
 winMoves.push\_back(m);  
 }  
 }  
 }  
 }  
 *//accumulating losing moves* if (moves\_o.size() > 2) {  
 for (int i = 0; i < moves\_o.size()-1; i++) {  
 for (int j = i+1; j < moves\_o.size(); j++) {  
 int m = 15 - (moves\_o[i] + moves\_o[j]);  
 if (m > 0 && m < 10) {  
 lossMoves.push\_back(m);  
 }  
 }  
 }  
 }  
 *//evaluating the accumulated moves  
 //check for wins* if (winMoves.size() > 0) {  
 for (int i = 0; i < winMoves.size(); i++) {  
 for (int j = 0; j < moves\_p.size(); j++) {  
 if (winMoves[i] == moves\_p[j]) {  
 return 1;  
 }  
 }  
 }  
 }  
 *//check for losses* if (lossMoves.size() > 0) {  
 for (int i = 0; i < lossMoves.size(); i++) {  
 for (int j = 0; j < moves\_o.size(); j++) {  
 if (lossMoves[i] == moves\_o[j]) {  
 return -1;  
 }  
 }  
 }  
 }  
 *//return 0 if no win or loss* return 0;  
}  
  
*// Yuvanesh evaluation  
// this function evaluated the board by count the win possibilities for 2 of the same pieces in a row/col/diagonal  
// and also for 1 piece in a row/col/diagonal  
// this is also static*int evaluate4(char board[3][3], char player, char opp){  
 int x2 =0;  
 int o2 =0;  
 int x1=0;  
 int o1 =0;  
  
 char opponent = opp;  
 for(int row =0; row < 3; row++){  
 *// checks for win possibilities for 2 pieces on board* if(board[row][0]== player && board[row][1]== player && board[row][2]!= (opponent||player) ){  
 x2++;  
 }  
 if(board[row][1]== player && board[row][2]== player && board[row][0]!= (opponent||player) ){  
 x2++;  
 }  
 if(board[row][0]== player && board[row][2]== player && board[row][1]!= (opponent||player) ){  
 x2++;  
 }  
 if(board[row][0]== opponent && board[row][1]== opponent && board[row][2]!= (opponent||player) ){  
 o2++;  
 }  
 if(board[row][1]== opponent && board[row][2]== opponent && board[row][0]!= (opponent||player) ){  
 o2++;  
 }  
 if(board[row][0]== opponent && board[row][2]== opponent && board[row][1]!= (opponent||player) ){  
 o2++;  
 }  
  
 *// checks for win possibilities for 1 piece on board* if(board[row][0]== player && board[row][1]!= (opponent||player) && board[row][2]!= (opponent||player) ){  
 x1++;  
 }  
 if(board[row][1]== player && board[row][0]!= (opponent||player) && board[row][2]!= (opponent||player) ) {  
 x1++;  
 }  
 if(board[row][2]== player && board[row][1]!= (opponent||player) && board[row][0]!= (opponent||player) ) {  
 x1++;  
 }  
 if(board[row][0]== opponent && board[row][1]!= (opponent||player) && board[row][2]!= (opponent||player) ){  
 o1++;  
 }  
 if(board[row][1]== opponent && board[row][0]!= (opponent||player) && board[row][2]!= (opponent||player) ) {  
 o1++;  
 }  
 if(board[row][2]== opponent && board[row][1]!= (opponent||player) && board[row][0]!= (opponent||player) ) {  
 o1++;  
 }  
 }  
 for(int col =0; col<3; col++){  
 *// checks for win possibilities for 2 pieces on board* if(board[0][col] == player && board[1][col]== player && board[2][col]!= (opponent||player) ){  
 x2++;  
 }  
 if(board[1][col] == player && board[2][col]== player && board[0][col]!= (opponent||player) ){  
 x2++;  
 }  
 if(board[2][col] == player && board[0][col]== player && board[1][col]!= (opponent||player) ){  
 x2++;  
 }  
 if(board[0][col] == opponent && board[1][col]== opponent && board[2][col]!= (opponent||player) ){  
 o2++;  
 }  
 if(board[0][col] == opponent && board[2][col]== opponent && board[1][col]!= (opponent||player) ){  
 o2++;  
 }  
 if(board[2][col] == opponent && board[1][col]== opponent && board[0][col]!= (opponent||player) ){  
 o2++;  
 }  
 *// checks for win possibilities for 1 piece on board* if(board[0][col] == player && board[1][col]!= (opponent||player) && board[2][col]!= (opponent||player) ){  
 x2++;  
 }  
 if(board[1][col] == player && board[2][col]!= (opponent||player) && board[1][col]!= (opponent||player) ){  
 x2++;  
 }  
 if(board[2][col] == player && board[1][col]!= (opponent||player) && board[0][col]!= (opponent||player) ){  
 x2++;  
 }  
 if(board[0][col] == opponent && board[1][col]!= (opponent||player) && board[2][col]!= (opponent||player) ){  
 o2++;  
 }  
 if(board[1][col] == opponent && board[2][col]!= (opponent||player) && board[0][col]!= (opponent||player) ){  
 o2++;  
 }  
 if(board[2][col] == opponent && board[1][col]!= (opponent||player) && board[0][col]!= (opponent||player) ){  
 o2++;  
 }  
 }  
 *// checks for win possibilities for 2 pieces on board* if(board[2][2] == player && board[1][1]== player && board[0][0]!= (opponent||player) ){  
 x2++;  
 }  
 if(board[0][0] == player && board[2][2]== player && board[1][1]!= (opponent||player) ){  
 x2++;  
 }  
 if(board[0][0] == player && board[1][1]== player && board[2][2]!= (opponent||player) ){  
 x2++;  
 }  
 if(board[2][2] == opponent && board[1][1]== opponent && board[0][0]!= (opponent||player) ){  
 o2++;  
 }  
 if(board[0][0] == opponent && board[2][2]== opponent && board[1][1]!= (opponent||player) ){  
 o2++;  
 }  
 if(board[0][0] == opponent && board[1][1]== opponent && board[2][2]!= (opponent||player) ){  
 o2++;  
 }  
 *//other diagonal* if(board[2][0] == player && board[1][1]== player && board[0][2]!= (opponent||player) ){  
 x2++;  
 }  
 if(board[0][2] == player && board[2][0]== player && board[1][1]!= (opponent||player) ){  
 x2++;  
 }  
 if(board[0][2] == player && board[1][1]== player && board[2][0]!= (opponent||player) ){  
 x2++;  
 }  
 if(board[2][0] == opponent && board[1][1]== opponent && board[0][2]!= (opponent||player) ){  
 o2++;  
 }  
 if(board[0][2] == opponent && board[2][0]== opponent && board[1][1]!= (opponent||player) ){  
 o2++;  
 }  
 if(board[0][2] == opponent && board[1][1]== opponent && board[2][0]!= (opponent||player) ){  
 o2++;  
 }  
  
 *// checks for win possibilities for 1 piece on board* if(board[2][2] == player && board[1][1]!= (opponent||player) && board[0][0]!= (opponent||player) ){  
 x1++;  
 }  
 if(board[0][0] == player && board[2][2]!= (opponent||player) && board[1][1]!= (opponent||player) ){  
 x1++;  
 }  
 if(board[1][1] == player && board[0][0]!= (opponent||player) && board[2][2]!= (opponent||player) ){  
 x1++;  
 }  
 if(board[2][2] == opponent && board[1][1]!= (opponent||player) && board[0][0]!= (opponent||player) ){  
 o1++;  
 }  
 if(board[0][0] == opponent && board[2][2]!= (opponent||player) && board[1][1]!= (opponent||player) ){  
 o1++;  
 }  
 if(board[1][1] == opponent && board[0][0]!= (opponent||player) && board[2][2]!= (opponent||player) ){  
 o1++;  
 }  
 *//other diagonal* if(board[2][0] == player && board[1][1]!= (opponent||player) && board[0][2]!= (opponent||player) ){  
 x1++;  
 }  
 if(board[0][2] == player && board[2][0]!= (opponent||player)&& board[1][1]!= (opponent||player) ){  
 x1++;  
 }  
 if(board[1][1] == player && board[0][2]!= (opponent||player) && board[2][0]!= (opponent||player) ){  
 x1++;  
 }  
 if(board[2][0] == opponent && board[1][1]!= (opponent||player) && board[0][2]!= (opponent||player) ){  
 o1++;  
 }  
 if(board[0][2] == opponent && board[2][0]!= (opponent||player) && board[1][1]!= (opponent||player) ){  
 o1++;  
 }  
 if(board[1][1] == opponent && board[0][2]!= (opponent||player) && board[2][0]!= (opponent||player) ){  
 o1++;  
 }  
 return 2\*x2+x1-2\*o2-o1;  
}  
  
  
  
int miniMax(char board[3][3], int depth, char player, int use, int pass){  
 int score;  
 int bestScore;  
 if(turn == 'X') {  
 if(!gameover()) { *// uncomment the return statement depending on which evaluation* return evaluate1(board, 'X','O');  
 *// return evaluate('X'); // not used in program runs  
 // return evaluate2(board,'X',depth);  
 // return evaluate3(board, 'X','O');  
 // return evaluate4(board,'X','O');* }  
 if (player == 'X') {  
 bestScore = -INT\_MAX;  
 for (int i = 0; i < 3; i++) {  
 for (int j = 0; j < 3; j++) {  
 if (board[i][j] != 'X' && board[i][j] != 'O') {  
 char temp = board[i][j];  
 board[i][j] = 'X';  
 score = miniMax(board, depth + 1, 'O', -pass, -use);  
 board[i][j] = temp;  
 allTries++;  
 if (score > bestScore) {  
 bestScore = score;  
 }  
 if (score > use) {  
 use = score;  
 }  
 if (pass <= use) {  
 break;  
 }  
 allTriesP++;  
 }  
 }  
 }  
 return bestScore;  
 } else {  
 bestScore = INT\_MAX;  
 for (int i = 0; i < 3; i++) {  
 for (int j = 0; j < 3; j++) {  
 if (board[i][j] != 'X' && board[i][j] != 'O') {  
 char temp = board[i][j];  
 board[i][j] = 'O';  
 score = miniMax(board, depth + 1, 'X', -use, -pass);  
 allTries++;  
 board[i][j] = temp;  
 if (score < bestScore) {  
 bestScore = score;  
 }  
 if (score < pass) {  
 use = score;  
 }  
 if (pass <= use) {  
 break;  
 }  
 allTriesP++;  
  
 }  
 }  
 }  
 return bestScore;  
 }  
 }else if (turn == 'O'){  
 if (!gameover()) {*// uncomment the return statement depending on which evaluation* return evaluate1(board, 'X','O');  
 *// return evaluate('O'); // Not used in runs  
 // return evaluate2(board,'O',depth);  
 // return evaluate3(board, 'X','O');  
 // return evaluate4(board,'X','O');* }  
 if (player == 'O') {  
 bestScore = -INT\_MAX;  
 for (int i = 0; i < 3; i++) {  
 for (int j = 0; j < 3; j++) {  
 if (board[i][j] != 'X' && board[i][j] != 'O') {  
 char temp = board[i][j];  
 board[i][j] = 'O';  
 score = miniMax(board, depth + 1, 'X', -pass, -use);  
 allTries++;  
 board[i][j] = temp;  
 if (score > bestScore) {  
 bestScore = score;  
 }  
 if (score > use) {  
 use = score;  
 }  
 if (pass <= use) {  
 break;  
 }  
 allTriesP++;  
  
 }  
 }  
 }  
 return bestScore;  
 } else {  
 bestScore = INT\_MAX;  
 for (int i = 0; i < 3; i++) {  
 for (int j = 0; j < 3; j++) {  
 if (board[i][j] != 'X' && board[i][j] != 'O') {  
 char temp = board[i][j];  
 board[i][j] = 'X';  
 score = miniMax(board, depth + 1, 'O', -use, -pass);  
 board[i][j] = temp;  
 allTries++;  
 if (score < bestScore) {  
 bestScore = score;  
 }  
 if (score < pass) {  
 use = score;  
 }  
 if (pass <= use) {  
 break;  
 }  
 allTriesP++;  
 }  
 }  
 }  
 return bestScore;  
 }  
 }  
}  
*//Function to get the game status e.g. GAME WON, GAME DRAW GAME IN CONTINUE MODE*bool gameover(){  
 *//checking the win for Simple Rows and Simple Column* if ((board[0][0] == 'X' && board[0][1] == 'X' && board[0][2] == 'X')  
 || (board[1][0] == 'X' && board[1][1] == 'X' && board[1][2] == 'X')  
 || (board[2][0] == 'X' && board[2][1] == 'X' && board[2][2] == 'X')) {  
 xWin = true;  
 return false;  
 } else if ((board[0][0] == 'O' && board[0][1] == 'O' && board[0][2] == 'O')  
 || (board[1][0] == 'O' && board[1][1] == 'O' && board[1][2] == 'O')  
 || (board[2][0] == 'O' && board[2][1] == 'O' && board[2][2] == 'O')) {  
 oWin = true;  
 return false;  
 }  
  
 if ((board[0][0] == 'X' && board[1][0] == 'X' && board[2][0] == 'X')  
 || (board[0][1] == 'X' && board[1][1] == 'X' && board[2][1] == 'X')  
 || (board[0][2] == 'X' && board[1][2] == 'X' && board[2][2] == 'X')) {  
 xWin = true;  
 return false;  
 } else if ((board[0][0] == 'O' && board[1][0] == 'O' && board[2][0] == 'O')  
 || (board[0][1] == 'O' && board[1][1] == 'O' && board[2][1] == 'O')  
 || (board[0][2] == 'O' && board[1][2] == 'O' && board[2][2] == 'O')) {  
 oWin = true;  
 return false;  
 }  
  
 *//checking the win for both diagonal* if ((board[0][0] == 'X' && board[1][1] == 'X' && board[2][2] == 'X')  
 || (board[0][2] == 'X' && board[1][1] == 'X' && board[2][0] == 'X')){  
 xWin = true;  
 return false;  
 } else if ((board[0][0] == 'O' && board[1][1] == 'O' && board[2][2] == 'O')  
 || (board[0][2] == 'O' && board[1][1] == 'O' && board[2][0] == 'O')) {  
 oWin = true;  
 return false;  
 }else{  
 int count = 0;  
 for(int i = 0; i<3;i++){  
 for(int j = 0; j < 3; j++){  
 if(board[i][j] == 'X' || board[i][j] == 'O')  
 count++;  
 }  
 }  
 if(count == 9) {  
 draw = true;  
 return false;  
 }  
 }  
 return true;  
}  
  
*//Program Main Method*int main()  
{  
 pid\_t pid;  
 pid = getpid();  
 cout<<"T I C K -- T A C -- T O E -- G A M E" << endl;  
 cout<<"FOR 2 PLAYERS" << endl;  
 int start\_s = clock();  
 while(gameover()){  
 display\_board();  
 player\_turn();  
 gameover();  
 }  
 int stop\_s = clock();  
 if(xWin){  
 cout<<"Congratulations!Player with 'X' has won the game" << endl;  
 display\_board();  
 }  
 else if(oWin){  
 cout<<"Congratulations!Player with 'O' has won the game" << endl;  
 display\_board();  
 }  
 else if(draw)  
 {  
 cout << "GAME DRAW!!!" << endl;  
 display\_board();  
 }  
  
 cout << "Total Nodes Generated for both players: " << totalTries << endl;  
 cout << "Total Nodes Generated for player X: " << totalTriesXbP << endl;  
 cout << "Total Nodes Generated for player O: " << totalTriesObP << endl;  
 cout << "Total memory used for player X: " << (totalTriesXbP\*9)/1000 << " kbs" << endl;  
 cout << "Total memory used for player O: " << (totalTriesObP\*9)/1000 << " kbs" << endl;  
  
 cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << endl;  
  
 cout << "Total Nodes Generated after Pruning for both players: " << totalTriesP << endl;  
 cout << "Total Nodes Expanded after Pruning for player X: " << totalTriesX << endl;  
 cout << "Total Nodes Expanded after Pruning for player O: " << totalTriesO << endl;  
 cout << "Total memory used after Pruning for player X: " << (totalTriesX\*9)/1000 << " kbs" << endl;  
 cout << "Total memory used after Pruning for player O: " << (totalTriesO\*9)/1000 << " kbs" << endl;  
 cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << endl;  
 *//cout << "Process ID for this program: " << pid << endl;* cout << "Program took " << (stop\_s-start\_s)/double(CLOCKS\_PER\_SEC)\*1000 << " ms\n";  
 cout << "Time for player X : " << totalTime1 << " ms" << endl;  
 cout << "Time for player O : " << totalTime2 << " ms" << endl;  
  
 *//system("ps u");* return 0;  
}

**Program Run:**

Max (Running MinMax A-B with EV#1) verses Min (Running MinMax A-B with EV #2):

T I C K -- T A C -- T O E -- G A M E

FOR 2 PLAYERS

PLAYER - 1 [X] PLAYER - 2 [O]

| |

1 | 2 | 3

\_\_\_\_\_|\_\_\_\_\_|\_\_\_\_\_

| |

4 | 5 | 6

\_\_\_\_\_|\_\_\_\_\_|\_\_\_\_\_

| |

7 | 8 | 9

| |

Player - 1 [X] turn : 0

53683 Nodes Generated

17142 Nodes Generated after Prune

PLAYER - 1 [X] PLAYER - 2 [O]

| |

X | 2 | 3

\_\_\_\_\_|\_\_\_\_\_|\_\_\_\_\_

| |

4 | 5 | 6

\_\_\_\_\_|\_\_\_\_\_|\_\_\_\_\_

| |

7 | 8 | 9

| |

Player - 2 [O] turn : 0

15729 Nodes Generated

8080 Nodes Generated after Prune

PLAYER - 1 [X] PLAYER - 2 [O]

| |

X | 2 | 3

\_\_\_\_\_|\_\_\_\_\_|\_\_\_\_\_

| |

4 | O | 6

\_\_\_\_\_|\_\_\_\_\_|\_\_\_\_\_

| |

7 | 8 | 9

| |

Player - 1 [X] turn : 0

2304 Nodes Generated

819 Nodes Generated after Prune

PLAYER - 1 [X] PLAYER - 2 [O]

| |

X | X | 3

\_\_\_\_\_|\_\_\_\_\_|\_\_\_\_\_

| |

4 | O | 6

\_\_\_\_\_|\_\_\_\_\_|\_\_\_\_\_

| |

7 | 8 | 9

| |

Player - 2 [O] turn : 0

498 Nodes Generated

195 Nodes Generated after Prune

PLAYER - 1 [X] PLAYER - 2 [O]

| |

X | X | O

\_\_\_\_\_|\_\_\_\_\_|\_\_\_\_\_

| |

4 | O | 6

\_\_\_\_\_|\_\_\_\_\_|\_\_\_\_\_

| |

7 | 8 | 9

| |

Player - 1 [X] turn : 0

106 Nodes Generated

39 Nodes Generated after Prune

PLAYER - 1 [X] PLAYER - 2 [O]

| |

X | X | O

\_\_\_\_\_|\_\_\_\_\_|\_\_\_\_\_

| |

4 | O | 6

\_\_\_\_\_|\_\_\_\_\_|\_\_\_\_\_

| |

X | 8 | 9

| |

Player - 2 [O] turn : 0

37 Nodes Generated

15 Nodes Generated after Prune

PLAYER - 1 [X] PLAYER - 2 [O]

| |

X | X | O

\_\_\_\_\_|\_\_\_\_\_|\_\_\_\_\_

| |

O | O | 6

\_\_\_\_\_|\_\_\_\_\_|\_\_\_\_\_

| |

X | 8 | 9

| |

Player - 1 [X] turn : 0

10 Nodes Generated

6 Nodes Generated after Prune

PLAYER - 1 [X] PLAYER - 2 [O]

| |

X | X | O

\_\_\_\_\_|\_\_\_\_\_|\_\_\_\_\_

| |

O | O | X

\_\_\_\_\_|\_\_\_\_\_|\_\_\_\_\_

| |

X | 8 | 9

| |

Player - 2 [O] turn : 0

2 Nodes Generated

2 Nodes Generated after Prune

PLAYER - 1 [X] PLAYER - 2 [O]

| |

X | X | O

\_\_\_\_\_|\_\_\_\_\_|\_\_\_\_\_

| |

O | O | X

\_\_\_\_\_|\_\_\_\_\_|\_\_\_\_\_

| |

X | O | 9

| |

Player - 1 [X] turn : 0

0 Nodes Generated

0 Nodes Generated after Prune

GAME DRAW!!!

PLAYER - 1 [X] PLAYER - 2 [O]

| |

X | X | O

\_\_\_\_\_|\_\_\_\_\_|\_\_\_\_\_

| |

O | O | X

\_\_\_\_\_|\_\_\_\_\_|\_\_\_\_\_

| |

X | O | X

| |

Total Nodes Generated for both players: 72369

Total Nodes Generated for player X: 56103

Total Nodes Generated for player O: 16266

Total memory used for player X: 504 kbs

Total memory used for player O: 146 kbs

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Total Nodes Generated after Pruning for both players: 26298

Total Nodes Expanded after Pruning for player X: 18006

Total Nodes Expanded after Pruning for player O: 8292

Total memory used after Pruning for player X: 162 kbs

Total memory used after Pruning for player O: 74 kbs

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Program took 12.518 ms

Time for player X : 9 ms

Time for player O : 1 ms

**Analysis of Program Run:**

The program run I have displayed above is the Max (Running MinMax A-B with EV#1) verses Min (Running MinMax A-B with EV #2) The result of the two is a tie. In the run you can see the features of our program. The program starts with an empty board and starts with player X. The main feature of this program is the game will continuously play until the game is over and you can see that it prints out each move a player makes. The program also prints out the nodes that are generated for each move. All of these features allows us to easily see the path of the game and just how many nodes the algorithm is visiting.

**Tabulation of Results:**

**Six Runs:**

**Data Analysis of Six Runs:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Max:EV1 | Max:EV2 | Max:EV3 | Min:EV2 | Min:EV3 | Min:EV4 |
| AVG Nodes Generated | 56062.66667 | 62010.5 | 38257 | 16266 | 9918.5 | 12029 |
| AVG Nodes Expanded | 17988.33333 | 23137 | 5411 | 8292 | 2162 | 4135.333333 |
| AVG execution time ms | 8 | 6 | 87 | 1 | 24.5 | 1.666666667 |
| AVG Memory Size kbs | 161.3333333 | 207.5 | 48 | 74 | 19 | 37 |
| Win percentage | 66.66666667 | 100 | 100 | 0 | 0 | 0 |

**Analysis of Results:**

The first main takeaway of the results is that Alpha-Beta pruning is hugely beneficial in a game of Tic-tac-toe. The nodes expanded across the board are way smaller than the total nodes generated. This is great for memory space and time. Another observation is that besides the first run, the Max player always won. This is probably why it seems like EV4 is a bad evaluation function since it was only ever the Min player. However, some evaluations were better than others such as EV3 that had the least amount of nodes expanded, smallest memory size, and a 100% win percentage. This data is skewed and the time worse than the other evaluations but it is still the better evaluation function.

**Conclusions:**

Working on this project has allowed me to further my understanding of the MinMax algorithm and deepen my skills on cpp.

**Team Member Contributions:**

Yuvanesh Rajamani:

* Worked on minMax algorithm
* Set up base evaluation
* Set up my own evaluation
* Helped set up program
* Helped program display proper information
* debugging

Robert Maldonado:

* Set up main program
* Set up minmax algorithm
* Developed two evaluation function
* Set up timer for program
* Set up memory size display
* debugging

John Edwards:

* Worked on minMax algorithm
* developed evaluation
* Helped set up program
* Helped set up timer
* Helped set up memory size
* debugging

**References:**

* Dr.Moonis Ali
* Online resources – youtube -Sebastian Lague